Parallel Mesh Processing

Dissertation

zur
Erlangung des Doktorgrades (Dr. rer. nat.)
des
Fachbereichs Mathematik und Informatik
der Philipps-Universität Marburg

vorgelegt von
Dipl.-Inform. Evgenij Derzapf
(geboren Mirnyj)

Marburg, Oktober 2012

Philipps-Universität Marburg
Fachbereich Mathematik und Informatik
Hans-Meerwein Straße 3, 35032 Marburg

Parallel Mesh Processing

Dissertation

zur
Erlangung des Doktorgrades (Dr. rer. nat.)
des
Fachbereichs Mathematik und Informatik
der Philipps-Universität Marburg

vorgelegt von
Dipl.-Inform. Evgenij Derzapf
(geboren Mirnyj)

Marburg, Oktober 2012

Philipps-Universität Marburg
Fachbereich Mathematik und Informatik
Hans-Meerwein Straße 3, 35032 Marburg

Angefertigt mit Genehmigung des Fachbereichs Mathematik und
Informatik der Philipps-Universität Marburg

Gutachter:
Prof. Dr. Michael Guthe, Philipps-Universität Marburg
Prof. Dr. Günther Greiner, Friedrich-Alexander-Universität Erlangen-Nürnberg

Prüfungskommission:
Prof. Dr. Michael Guthe, Philipps-Universität Marburg
Prof. Dr. Günther Greiner, Friedrich-Alexander-Universität Erlangen-Nürnberg
Prof. Dr. Manfred Sommer, Philipps-Universität Marburg
Prof. Dr. Rita Loogen, Philipps-Universität Marburg
Prof. Dr. Ekaterina Kostina, Philipps-Universität Marburg

Abgabetermin: 23. August 2012
Prüfungstermin: 11. Oktober 2012

To my wife Diana,
my parents Irene and Johann
and my sister Viktoria.

Acknowledgments

The presented work has been produced within the scope of the *AG Grafik und Multimedia* at the Institute of Computer Science of the University Marburg. At this point I like to thank all people who were directly or indirectly involved in the creation of this work.

My thanks belong primarily to Prof. Michael Guthe without whose ideas, many discussions, and support this work would not possible.

I thank Prof. Manfred Sommer, my second adviser and all members of the group for useful advices. Especially I also thank Nicolas Menzel, Nico Grund, and Christine Ulrich for the joint publications.

Additionally I thank the *Stanford 3D Scanning Repository*, *The volume library*, *Turbosquid*, *Eurographics*, *Crytek*, and the *Digital Michelangelo Project* for the models used in this thesis.

Zusammenfassung

Die aktuelle Forschung im Bereich der Computergrafik versucht den zunehmenden Ansprüchen der Anwender gerecht zu werden und erzeugt immer realistischer wirkende Bilder. Dementsprechend werden die Szenen und Verfahren, die zur Darstellung der Bilder genutzt werden, immer komplexer. So eine Entwicklung ist unweigerlich mit der Steigerung der erforderlichen Rechenleistung verbunden, da die Modelle, aus denen eine Szene besteht, aus Milliarden von Polygonen bestehen können und in Echtzeit dargestellt werden müssen.

Die realistische Bilddarstellung ruht auf drei Säulen: Modelle, Materialien und Beleuchtung. Heutzutage gibt es einige Verfahren für effiziente und realistische Approximation der globalen Beleuchtung. Genauso existieren Algorithmen zur Erstellung von realistischen Materialien. Es gibt zwar auch Verfahren für das Rendering von Modellen in Echtzeit, diese funktionieren aber meist nur für Szenen mittlerer Komplexität und scheitern bei sehr komplexen Szenen.

Die Modelle bilden die Grundlage einer Szene; deren Optimierung hat unmittelbare Auswirkungen auf die Effizienz der Verfahren zur Materialdarstellung und Beleuchtung, so dass erst eine optimierte Modellrepräsentation eine Echtzeitdarstellung ermöglicht.

Viele der in der Computergrafik verwendeten Modelle werden mit Hilfe der Dreiecksnetze repräsentiert. Das darin enthaltende Datenvolumen ist enorm, um letztlich den Detailreichtum der jeweiligen Objekte darstellen bzw. den wachsenden Realitätsanspruch bewältigen zu können. Das Rendern von komplexen, aus Millionen von Dreiecken bestehenden Modellen stellt selbst für moderne Grafikkarten eine große Herausforderung dar. Daher ist es insbesondere für die Echtzeitsimulationen notwendig, effiziente Algorithmen zu entwickeln. Solche Algorithmen sollten einerseits Visibility Culling[1], Level-of-Detail (LOD)[2], Out-of-Core Speicherverwaltung[3] und Kompression unterstützen. Anderseits sollte diese Optimierung sehr effizient arbeiten, um das Rendering nicht noch zusätzlich zu behindern. Dies erfordert die Entwicklung paralleler Verfahren, die in der Lage sind, die enorme Datenflut effizient zu verarbeiten.

Der Kernbeitrag dieser Arbeit sind neuartige Algorithmen und Datenstrukturen, die speziell für eine effiziente parallele Datenverarbeitung entwickelt wurden und in der Lage

1 Entfernen nicht sichtbarer Geometrie.
2 Repräsentation der Modell durch verschiedene Detailstufen.
3 Auslagerung der Daten auf externe Datenträger.

sind sehr komplexe Modelle und Szenen in Echtzeit darzustellen, sowie zu modellieren. Diese Algorithmen arbeiten in zwei Phasen: Zunächst wird in einer Offline-Phase[4] die Datenstruktur erzeugt und für parallele Verarbeitung optimiert. Die optimierte Datenstruktur wird dann in der zweiten Phase für das Echtzeitrendering verwendet.

Ein weiterer Beitrag dieser Arbeit ist ein Algorithmus, welcher in der Lage ist, einen sehr realistisch wirkenden Planeten prozedural zu generieren und in Echtzeit zu rendern.

4 Vorverarbeitungsschritt

Abstract

Current research in the field of computer graphics tries to meet the increasing user requirements and produces more realistic looking images. Accordingly, the scenes and techniques that are used to display the images tend to be come more complex. This increases the required computing power, because the models of the scene can consist of billions of polygons and must be displayed in real-time.

Realistic image rendering consist of three parts: models, materials and illumination. Today, some methods for efficient and realistic approximation of the global illumination exist. Similarly, there are algorithms for the creation of realistic materials. There are also methods for real-time rendering of models available, but most of them work only for scenes of average complexity and fail for very complex scenes.

The models are the base of a scene. The optimization of the model representation has a direct impact on the efficiency of algorithms for rendering of material and illumination, so that only an optimized model representation enables real-time rendering.

Many models used in computer graphics are represented by triangle meshes. The amount of data is mostly enormous, to represent the richness of detail and meet the increasing requirements for real-time rendering. Despite the enormous processing power of modern GPUs, highly detailed models cannot be rendered in real-time. Therefore, particularly with regard to real-time simulations, it is necessary to develop efficient algorithms. These algorithms need to be used for visibility culling[5], level-of-detail (LOD)[6], out-of-core memory management[7], and compression. On the other hand, these algorithms should work very efficiently, to avoid impeding the rendering. Parallel data processing is required to efficiently process large amounts of data.

The core contributions of this thesis are new algorithms and data structures which are specially developed for efficient parallel data processing. They allow for real time rendering and editing of very complex models and scenes. These algorithms are divided into two phases: First, in an offline-phase[8], data structures are generated and optimized for parallel processing. In the second phase, the generated data structures are used for

5 Removing of non-visible geometry.
6 Different levels of detail of the models.
7 Swapping the data on external devices.
8 Preprocessing step

real time rendering.

Another contribution of this thesis is an algorithm which generates procedurally a very realistic looking planet and render it in real-time.

Vorwort

Falls Gott die Welt geschaffen hat, war seine Hauptsorge sicher nicht, sie so zu machen, dass wir sie verstehen können.

Albert Einstein

Ich kann mich noch gut daran erinnern, als ich Anfang der 90er, meinen ersten Computer gesehen habe. Ich muss 8 oder 9 Jahre alt gewesen sein. Zusammen mit meinem damaligen Kumpel saß ich unerlaubt vor dem Computer seines Vaters. Dieses Gebilde, aus Metall und Plastik auf wundersame Weise verbunden, schien zu leben. Aber wie Kinder nun einmal so sind, hatten wir nur ein Gedanken, welcher kein anderer sein konnte, als ein Computerspiel zu spielen. Lange haben wir uns durch das Menü kämpfen müssen, geblendet durch den grellen blauen Hintergrund. Doch dann war es geschafft. Endlich, nach minutenlangen Rattern und Knirschen startete das Spiel. Eine Echtzeit-Kampfflugsimulation hat uns in ihren Bann gezogen. Diese faszinierende 3D-Grafik, verpixelt, aber dennoch ein Zeugnis von menschlichem Streben nach Vollkommenheit, hat uns für mehrere Stunden in eine andere Welt entführt.

Jetzt, viele Jahre später, hat sich vieles geändert, Eines blieb: Der menschliche Wille die Welt zu verstehen. So hoffe ich mit meiner Arbeit einen entsprechenden Beitrag geleistet und mit diesem die Menschheit jenem Ziel ein Stückchen näher gebracht zu haben.

Contents

CHAPTER 1

Introduction

Today, high quality simulations of the real world, called virtual reality, can be found in many aspects of the daily life (e.g. films, computer games and medical applications). The user requirements increase and therefore the virtual reality becomes more complex.

To satisfy the ever growing demand for realistic images, the complexity of polygonal models constantly increases. Possible methods for generating such models are procedural creation[9], 3D laser scanner, medical devices (e.g. Computer Tomograph (CT) or Magnetic Resonance Tomograph (MRT)) or geological scans. Most interesting are the procedurally generated terrains or even complete planets. Such planets can be generated without user. They can contain realistic mountains, seas, rivers, atmosphere, climate zones and vegetation. The rivers are one of the most important parts of terrain, since they are vital for life and can be important for navigation. In particular, the real-time generation of realistic rivers networks is a major problem. The realistic rivers can branch, have different depth and slope. Moreover, the river bed and course need to satisfy a certain degree of realism. Additionally, it is important to have a possibility for editing of the generated or scanned models. Such real-time editing algorithms can be used for post editing or animation.

The constantly increasing complexity of polygonal models in interactive applications imposes two major problems. First, the number of primitives that can be rendered at real-time frame rates is currently limited to a few million. Second, less than 45 million triangles[10] - with vertices and normal - can be stored per gigabyte. Additionally, the editing of the the generated or scanned models is very problematic, because of the high number of vertices and high memory consumption.

Despite the enormous processing power of the GPU (Graphics Processing Unit), such models can not be rendered, generated or edited in real-time. While the rendering time can be reduced using level-of-detail (LOD) algorithms, representing a model at

9 Using mathematical functions.
10 Using 32 bit floats.

different complexity levels[11] , these often even increase memory consumption. Out-of-core algorithms solve this problem by transferring the data currently required for rendering from external devices. Compression techniques are commonly used in this context, because of the bandwidth limitations. To achieve real-time frame rates, parallel data processing is required. For this, new data structures, which are suitable for parallel processing and real-time decompression, need to be developed. Such data structures are mostly created in one preprocessing simplification step, which can be highly complex and thus can take several hours, days or even months. Therefore, parallel simplification algorithms need to be developed, to reduce the preprocessing time.

To conform all these requirements, this thesis is focused on parallel simplification, rendering, editing and procedural generation of very large 3D models, with sizes of several gigabytes. To achieve the real-time frame rates, all of the data structures and algorithms presented in this thesis, are specially developed for acceleration through parallel processing. All algorithms use CUDA (Compute Unified Device Architecture) for parallel processing to exploit the advantage of GPGPU (General Purpose Compution on Graphics Processing Unit). Because almost every modern computer offers hardware accelerated graphics, no special hardware is required. The inexpensive customary NVIDIA GPU[12] is enough for execution of algorithms developed in this thesis.

NVIDIA developed the new programming language CUDA specifically for GPGPU computing to allow taking the advantage of enormous parallel computational power of the GPU for acceleration of non-graphical algorithms. By using CUDA for parallel data processing, these new techniques can be developed for real-time rendering, generation and editing of complex models.

The remainder of this thesis is structured as follows: Chapter 2 gives an brief overview of the theoretical basis used in this thesis and the NVIDIA CUDA programing technology. The remaining chapters present the algorithms which are developed in this thesis. Chapter 3 describes a technique for parallel static LOD mesh simplification. In chapter 4 a combination of a parallel out-of-core marching cubes implementation with a parallel stream simplification algorithm is presented. In chapter 5, progressive meshes based techniques for real-time in-core and out-of-core view-dependent mesh rendering of gigabyte-sized models are proposed. Chapter 6 describes a technique for parallel mesh editing. In chapter 7 procedural planet generation with realistic river networks is discuss. Finally, the chapter 8 gives an brief overview of the algorithms proposed in this thesis. Additionally, the possible improvements are presented.

11 Defined by the number of triangles.
12 Low cost consumer market NVIDIA GPU.

CHAPTER 2

Basics

This chapter is structured as follows: In section 2.1 the techniques for rendering complex (gigabyte-sized) triangle mesh models are briefly discussed. Section 2.2 gives a brief overview of the mesh extraction from volume data. In section 2.3 procedural mesh generation is briefly discussed. Finally, in section 2.4 Compute Unified Device Architecture (CUDA) is presented.

2.1 Complex Meshes

The graphics processors (GPUs) are constructed for rendering triangle (face) mesh models, which are discrete approximations of the model surface. The quality of the approximation depends on the number of triangles. The triangle mesh is an undirected graph with a set of vertices and edges. Two vertices define an edge and three edges or three vertices a face [CH91].

The need for high quality triangle models in interactive applications is constantly increasing. Despite the enormous processing power of the GPU, highly detailed models cannot be rendered in real-time. Often they even do not fit into graphics memory since a scene may contain several of them. To solve this problem, the number of triangles must be reduced using techniques such as visibility culling and levels of detail (LOD). Compressed data representations and out-of-core techniques are often used to further reduce the data size. Parallel data processing must be used to achieve real-time frame rates.

The remainder of this section is structured as follows: In section 2.1.1 visibility culling of triangle mesh models is discussed. Section 2.1.2 gives an brief overview of the levels of detail techniques. Finally, in section 2.1.3 and 2.1.4 memory management and data compression techniques are briefly discussed.

2.1.1 Visibility Culling

Visibility culling techniques try to determine the visible and invisible parts of a scene, in order to remove the invisible parts. View frustum culling removes triangles outside the view frustum. Triangles on the backside of an object are hidden by the objects front side. Backface culling takes advantage of this by testing if the normal of a triangle faces away from the position of the viewer. Finally, occlusion culling is used to remove all occluded triangles. For efficient occlusion culling the scene is typically partitioned using a spatial hierarchy.

2.1.2 Level of Detail

To reduce the number of triangles, existing techniques either use static or dynamic levels of detail (LODs). Static LODs represent each model at several resolutions. During rendering, a level is chosen per object based e.g. on the distance to the viewer (see Figure 2.1). Static LODs are easy to use since simplification and rendering are decoupled, but suffer from popping artifacts[13] if the simplification levels are not similar enough. In addition, the memory overhead compared to an ordinary mesh is typically about 50%. Dynamic LODs depart from this approach by encoding the simplification operations into a continuous sequence. The simplification algorithm often generates the simplified models by collapsing the edges. All data for the reconstruction operation is stored in advance. The edge collapse operation contracts an edge and the connecting vertices into a one collapse vertex and remove degenerated faces. As a result, popping artifacts are often nearly invisible. As the LOD is view-dependent[14], it uses no more faces than necessary. Another advantage is that back-facing polygons don't have to be refined, resulting in an overall reduction by a factor of three to four compared to static LODs[15].

2.1.3 Memory Management

The limitation of most level of detail algorithms is in-core memory management[16], i.e. the whole scene must fit into graphics memory. This constraint does not allow for rendering huge scenes exceeding the amount of available graphics memory. The memory problem is often solved by partitioning the scene using a spatial hierarchy and generating a level of detail for each node. Then out-of-core memory management is used by loading only currently required nodes in the GPU memory from external devices.

13 Visible transitions between two quality levels.
14 i.e. visibility culling techniques are used
15 For average scenes.
16 Existing algorithms for managing the level of detail are often limited to data sets that fit into main memory. This is called in-core memory management. Algorithms that work with data that does not fit into memory are classified as out-of-core.

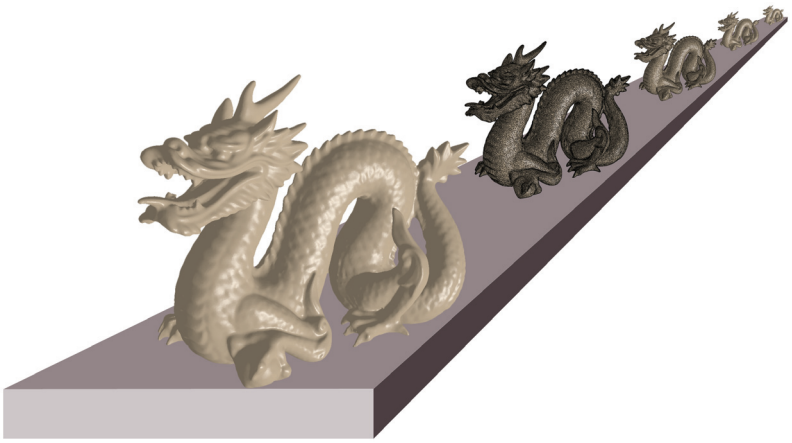

Figure 2.1: With increasing distance, coarser approximations of the triangle mesh model can be used, without affecting the visual appearance .

2.1.4 Compression

The problem of static and dynamic LOD techniques is that required memory is significantly higher as of the original model. For this reason, compressed representations of the data are often used to reduce the memory consumption and the required bandwidth for out-of-core memory management. To achieve real-time frame rates, the compression algorithm needs to have high compression rates and support high efficient decompression.

In the remainder of this section the different types of the coding algorithms are briefly discussed.

2.1.4.1 Fixed-to-Fixed

The fixed-to-fixed compression technique reduce a large set of n-bit words to a smaller set of m-bit codewords[17], where $m < n$.

Vector quantization is a lossy fixed-to-fixed compression technique. The vector data is divided in groups (clusters) that are approximately similar by using clustering algorithms (e.g. k-means) to determine the centroid of the cluster[18]. Then all centroids build a database and the data is encoded by the database index of the closest centroid. The decompression performance of the vector quantization is very high, but compression is lossy. Therefore, the usage for compressing the data that needs to be reconstructed exactly is only only limited, e.g. vertex position or normal, but for other values like colours.

17 e.g. 32-bit integer to 8-bit char.
18 i.e. there exists no injective transformation.

2.1.4.2 Fixed-to-Variable

Fixed-to-variable compression techniques encode n-bit words to codewords with the variable length.

The Huffman algorithm [Huf52] is a lossless fixed-to-variable data compression technique. The data is encoded using the frequency of occurrence of the strings. Strings with high occurrence are encoded using few bits and strings with low occurrence are encoded using more bits. The goal is to reduce the total length of the encoded sequence. The codewords are determined by constructing a Huffman tree (see [Huf52]). Left and right subtrees are labeled with 0 and 1, and traversing the tree yields the encoded string for a symbol.

The other lossless fixed-to-variable data compression technique is range arithmetic coding [Mar79, WNC87]. Unlike Huffman coding, the optimal number of bits is used for each codeword. The data are encoded by assigning an interval to each string. The algorithm begins with the interval [0..1), which is divided in subintervals. The size of the subintervals is dependent on the probability of the values, stored in the probability table. The algorithm starts with the first value of the string and selects the subinterval accordingly to the probability table. The interval for the next value is the subinterval of the current one and so on. The codewords are determined by choosing the subinterval of the last value. Context adaptive arithmetic coding is another form of arithmetic coding. It use different probability tables to achieve higher compression rates. The decompression performance of range arithmetic coding is low, but the compression rate is high.

2.1.4.3 Variable-to-Fixed

Variable-to-fixed compression techniques encode words of variable length to m-bit codewords.

Run-length coding [JT98] is a lossless variable-to-fixed data compression technique. The data is encoded as sequence of consecutive 0 or 1. The decompression performance is very high, but the compression rate is only high if the number of 0 and 1 is highly unbalanced.

Simple9 [AM04] is a variable-to-fixed data compression technique that is specifically developed for highly efficient decoding. The algorithm tries to pack as much integers as possible into 32 bits. For this, the 32 bits are split in 4 control and 28 data bits. On this way, the data bits can be divided in 9 ways. The possible cases are: 28 1-bit numbers, 14 2-bit numbers, 7 4-bit numbers, 5 5-bit numbers, 4 7-bit numbers, 3 9-bit numbers, 2 14-bit numbers and 1 28-bit number. The control bits are used to determine which of the cases is used. Later, the authors of Simple9 improve the compression rate by proposing similar algorithms called Relate10, Carryover12 and Simple16.

The other lossless variable-to-fixed data compression technique is LZW [KWP*03]. It uses a dictionary, that contains a frequently occurring strings. The new dictionary entries are generated by adding the next character to the already existing entry. Then the data is encoded by storing the dictionary index with a fixed size. The dictionary is encoded

implicitly into the data and does not need to be stored.

2.1.4.4 Variable-to-Variable

Variable-to-variable compression techniques encode words of variable length to codewords of variable length.

Compression techniques of this type are a combination of variable-to-fixed and fixed-to-variable compression. Bzip2 is a variable-to-variable compression technique which combines run-length, Burrows Wheeler Transform (BWT) algorithm [Man99] and Huffman coding. BWT permutes the order of the input characters. After permutation the output word contains many repeated characters and allows for higher compression rate.

2.1.4.5 Discussion

The algorithms proposed in this thesis use massive parallel data processing to achieve real-time frame rates. Because of fine granularity of the data, the number of parallel threads is high. Therefore, the used compression techniques need to support high decompression performance and high compression rates for data blocks of less than 1 KByte. For this reason, the best choice are the fixed-to-variable compression techniques. For these requirements, the context adaptive arithmetic coding has the best compression rates [ZLS08].

2.2 Mesh Extraction

Medical devices, e.g. CT or MRT, and geological scans produce data in form of a 3D Cartesian discretized scalar field (volume data set), often called voxel grid, to approximate the surface of an object. Such volume data sets have higher memory consumption than the triangle meshes and the visualisation takes significantly longer. Additionally, the editing is very complex or even not possible.

To reduce the rendering time and memory consumption, the triangle mesh is often extracted from volume data set. Such extraction algorithms are based on selecting an iso-value for the density and generating a triangle mesh that approximates the corresponding iso-surface. For the mesh extraction, the marching cubes algorithm (Section 2.2.1) is often used.

Due to the progress made in medical and geological imaging systems during the last decades, the resolution of volume data sets has constantly grown. This also increased the memory requirements and computation times. The number of triangles generated by iso-surface extraction algorithms has also grown. Therefore, the extracted triangle meshes need to be rendered using known algorithms (Section 2.1) to reduce the number of faces and memory consumption.

2.2.1 Marching Cubes Algorithm

The classical marching cubes algorithm [LC87] is based on a linear interpolation of the density between the vertices of the volume. The algorithm divides the data into cubes and processes them sequentially. The cubes are defined by rectilinear voxel grid, produced by devices (e.g. CT or MRT).

In every cube, the scalar values are classified using a user-defined iso-value. To approximate the surface of an object, the algorithm determines the intersection of the surface with the cubes. For this, a bit is assigned for each of eight vertices of the cube. The vertex is marked with 1, if the value of the vertex is higher or equals to the user-defined iso-value and 0 else. Then, all vertices marked with 1 are inside the object and the vertices marked with 0 outside. Accordingly, only $2^8 = 256$ cases[19] for the intersection of the object and cube are possible. All possible cases are stored in a lookup-table and used at runtime, to determine the intersected edges of the cube and triangles that are required for the surface approximation (some of cases are shown in the Figure 2.2). The intersections of the object and the edges of the cube are calculated using linear interpolation of the density between the vertices of the current edge. Finally, the gradients of the cube vertices are estimated to compute the normals for each vertex of the generated triangles.

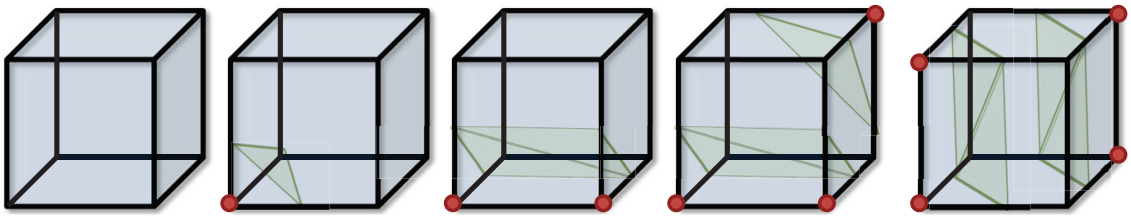

Figure 2.2: Five of 256 possible cases from the lookup-table. The vertices inside the object are marked in red and triangles required for the surface approximation in green [LC87].

Because every edge is shared by several cubes, the calculated intersections can be reused. Therefore, only three intersections need to be calculated for the cubes inside the grid. This reduces the computational time and memory consumption.

2.3 Procedural Algorithms

Rendering natural objects is a great challenge for todays computer graphics. Early, before Mandelbrot [Man83] found the high self-similarity grade of the natural objects (e.g. landscapes, trees or clouds), the formal definition of such objects with high richness of detail was not possible. Using fractals, such objects can be easy defined and automatical created using procedural algorithms. Accordingly to Mandelbrot [Man83] the fractals are defined as:

19 8 vertices, with two possible cases per vertex.

A fractal is by definition a set for which the Hausdorff Besicovitch dimension strictly exceeds the topological dimension.

The fractals are often defined recursively(see Figure 2.3). In this way, the visual richness of the object increases.

In each iteration, the number of copies N is given by the scaling factor ε and fractal dimension D:

$$N = \frac{1}{\varepsilon^D}, \tag{2.1}$$

The fractal dimension[20] give the measure for the complexity of the self-similar object. In a broader sense, it is a measure for the spatial extension and is defined as:

$$D = \frac{\log(N)}{\log(\varepsilon)}, \tag{2.2}$$

For example, the Sierpinski triangle (see Figure 2.3) consist of $N = 3$ copies with a scaling factor of $\varepsilon = 2$. The fractal dimension is calculated using equation 2.2 as:

$$D = \frac{\log(3)}{\log(2)} = 1.58. \tag{2.3}$$

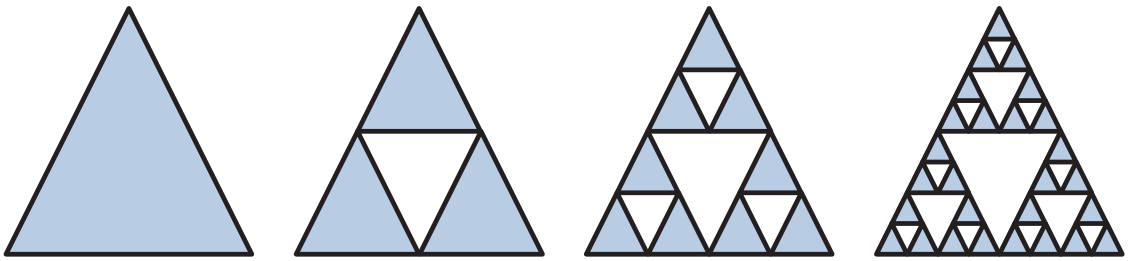

Figure 2.3: Construction of the Sierpinski triangle.

2.3.1 Fractional Brownian Motion

In a early 18th century Brown analyzed the motion of the particles (Brownian motion) [Bro28] in a fluid as a result of collisions between the particles. Later, the movement could be exactly described mathematically with particle theory and served as basis for creation of fractal objects (e.g. landscapes). Mandelbrot [MVN68] expands on Brownian motion with a Hurst exponent $H \in (0,1)$, to simulate a Brownian motion of a dimension $D = 2 - H$ (fBm – fractional Brownian motion) and determines a fractal dimension of 1.2 for the silhouette of a mountain. To define a landscapes the fBm need to be expand to 2D by overlap of several 1D fBms. The Hurst exponent is used to control the fractal

20 Also known as Hausdorff dimension or Hausdorff-Besicovitch dimension.

dimension, which is defined as:

$$D = E + 1 - H, \tag{2.4}$$

where E is the euclid dimension. With increasing H the created fractal objects obtain a rougher surface.

The fractional Brownian motion is a basic technique for landscape creation. Currently, three main techniques are used to generate fractal landscapes: Fourier Filtering, Noise Synthesis and Midpoint Displacement. Fourier filtering approximates the fBm using 2D inverse fourier transformation of Gaussian white noise function. The quality of the generated landscape is high, but the level of detail is fixed. Noise Synthesis approximate the fBm using continued noise function [Per85, Per02]. Midpoint displacement recursively inserts new vertices which split polygons into several new polygons. A new vertex's altitude is the average altitude of the surrounding vertices. Midpoint displacement and other techniques for landscape definition are discussed in section 7.1.

2.4 CUDA

Almost every modern computer offers hardware accelerated graphics, used by graphics applications. But this enormous computational power can also be used for acceleration of non-graphical applications (e.g. scientific simulations and mesh simplification).

Early[21], graphics hardware was hardwired for specific rendering algorithms, reprogramming was not possible or very difficult using low-level graphic assembly language or register combiners. Faced with increasing parallel computing power, GPU (Graphics Processing Unit) vendors decided to execute code written for the CPU (Central Processing Unit) on the GPU. The next logical step of the GPU progress was programmable hardware. The first language for GPU programming was a shading language (e.g. NVIDIA Cg and HLSL), but the programming possibilities were limited. As advancement of the shading languages, a new language CUDA (Compute Unified Device Architecture) [NVI11b] was presented by NVIDIA in 2007. CUDA was developed for GPGPU (General Purpose Compution on Graphics Processing Unit) computing, to allow users to take advantage of enormous parallel computational power of the GPU for acceleration of non-graphical algorithms.

The remainder of this section is structured as follows: Section 2.4.1 analyzes parallel performance and flexibility of CPU and GPU. Section 2.4.2 and section 2.4.3 give a brief overview of the CUDA hardware and software layer. Finally, the optimization possibilities are briefly discussed in section 2.4.4.

21 Early nineties.

2.4.1 CPU vs. GPU

Roughly between 2000 and 2003, the CPU market perceived a battle between Intel and AMD. This was reflected in rapidly rising CPU frequencies and associated increases of the CPU computing power. But then it came to an abrupt end, because of increasing production of thermal energy. The performance of the new CPUs was not more increased by the frequency, but by launching of the Multi-core CPUs [22]. Frequencies increased only slightly in the next few years and therefore the increase of the CPU performance was rather poor (Figure 2.4). The reason for it is, that the number of transistors increases exponentially with increasing CPU frequencies. Therefore, vast amounts of transistors were necessary to get only a few additional MHz. This increase the production of thermal energy drastically.

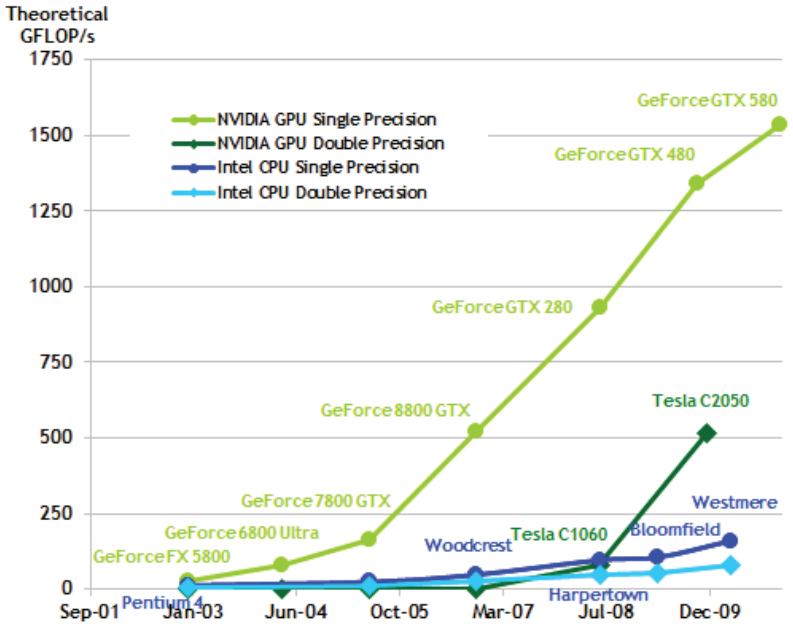

Figure 2.4: Comparison of the CPU and GPU performance (Floating-Point Operations per Second (FLOP/s)) of the last years [NVI11b].

In comparison to the CPU, the GPU computational power increase in the last years was enormous. During 2003 it was approximately equal to the CPU and increased in 2010 by about 10 times (Figure 2.4). The reason for this is that the single-core and multi-core CPUs are designed to produce maximal performance from a stream of instructions. These instructions can operate on different data, access memory randomly and run on divergent paths. Possibilities of parallel execution of instruction are very limited, because the CPUs are optimized for a sequential stream of instructions. This makes blindly increasing calculating units and cores useless.

22 Started by AMD.

The CPU manufacturers attempted to solve this problem using technologies such Hyper-Threading, SSE (Intel's extension of SIMD) and reordering of the instructions, but with moderate success. In contrast to the CPU, the job of the GPU is to handle a set of pixels and polygons, which can be executed independently. Therefore, the GPU is designed much more consequently for parallel execution and the performance benefits from increasing number of calculating units and cores. Additionally, the memory access of the GPU is extremely coherent, because of coherency of data (e.g. pixel and polygons). Therefore, the memory bandwidth corresponds nearly to the theoretical bandwidth. This reduces the size of required cache.

The CPU manufacturers are trying to improve parallelism in the execution of the instructions, while the GPU manufacturers attempt to organize GPUs more flexibly. CUDA offers nearly the flexibility of an ordinary programming language and allows to rewrite algorithms for execution on the GPU. But note, to achieve a full performance, the algorithm must be able to run in parallel without or with only few race conditions[23]. Additionally, the algorithms and data structures need to be optimized for parallel execution.

2.4.2 CUDA-Hardware Layer

NVIDIA designed a new GPU architecture specially for CUDA. The most important improvement was the fusion of the vertex and fragment processor to only one Streaming Processor (SP). The new processors were able to execute both tasks and increased the efficiency of the GPU. A new GPU consists of a set of SPs, which are grouped into Streaming Multiprocessors (SMs) and the SMs are grouped into Graphics Processor Clusters (GPCs)[24](Figure 2.5). All new GPUs from NVIDIA use this architecture. The number of GPCs per GPU, SMs per GPC and SPs per SM varies depending on the GPU model and determines the performance of the GPU.

In the CUDA hardware architecture, a distinction is drawn between two memory spaces:

Device Memory: A very large memory of DDRX SDRAM architecture[25]. All SPs of the GPU may access data from this memory, but the access latency is very high.

On-Chip Memory: Each SM has its own on-chip memory, only SPs of the same SM may access data from this memory. The on-chip memory consists of constant cache, texture cache, shared memory and registers. This memory is very small, but the access latency is very low.

The GPCs use MIMD architecture (Multiple Instruction Multiple Data) and the SMs SIMT architecture (Single Instruction Multiple Threads). NVIDIA designed the SIMT

23 To deal with the race conditions, atomic operations are in CUDA available, but this reduce the performance drastically.

24 GTX 580 GPU consists of 4 GPSs, 16 SMs and 512 SPs. Each GPC consists of 4 SMs and each SM consists of 32 SPs (Figure 2.5).

25 DDR5 SDRAM (Double Data Rate Synchronous Dynamic Random-Access Memory) architecture on the GTX 580 GPU, with size of 1.5 GB or 3.0 GB.

Figure 2.5: NVIDA GF100/GF110 (alias GTX480/GTX580) GPU core [NVI10].

architecture specially for CUDA. SIMT architecture is akin to the SIMD, but more flexible, e.g. the thread divergence is handled automatically by the hardware. Each SM executes groups of 32 threads in parallel, so called warps[26]. Therefore, the number of warps that can be executed in parallel depends on the number of SMs in the GPU. All threads in a warp start with the same instruction, but each thread has its own instruction count and can execute divergently on their own SP of the SM. Only these threads inside a warp that have the same instruction count can be executed in parallel. If the instruction count is different, the threads of the warp with the same count are grouped and the groups are executed sequentially on the SM. In the worst case, if all threads of the warp have different instruction counts, the execution is fully sequentially. Furthermore, the warps are grouped to the blocks. The warps of the same block are executed on the same SM. Then, the blocks are grouped in to the grid and can be executed on different GPCs in parallel. The grid can be 1D and 2D and the block 1D, 2D and 3D. The maximal number of threads per block and grid is limited by the GPU architecture[27] and is defined by the programmer. The decomposition of blocks to warps and the distribution of the resources is hardware supervised and cannot be controlled from programmer.

26 Each SM of the GTX 580 GPU consists 32 SPs and can execute all treads of the warp in parallel. E.g. the SM of the older GPU architecture GT200 (alias GTX285) consists 8 SPs, therefore only 8 threads (alias quarter-warp) of the warp can be executed in parallel.

27 On the GTX 580 the maximal number of threads per block is 1024 and the dimension is limited to maximum of $1024 \times 1024 \times 64$ threads. The grid dimension is limited to maximum of $65535 \times 65535 \times 1$ threads.

On the grid, block and thread layers, different memory spaces are available. The following list shows memory spaces on the grid layer (all threads of the same grid may access data from this memory):

Global Memory: A large cached device memory[28] with high latency.

Constant Memory: A small read only cached device memory[29] with low latency.

Texture Memory: A large read only cached device memory. It is optimized for data filtering and different addressing modes.

Memory available on the block layer (all threads of the same block may access data from this memory):

Shared Memory: A small on-chip memory[30] with low latency. It can be used for thread communication inside the block. The communication between the blocks through shared memory is not possible, because the blocks can be executed on different SMs.

Memory available on the thread layer (the access is exclusive for each thread):

Register: A small on-chip memory[31] with very low latency.

Local Memory: A large cached part of device memory.

The distribution of register and local memory is supervised by the compiler and cannot be controlled by programmer.

2.4.3 CUDA-Software Layer

CUDA allows to define host code, which is execute sequentially on the CPU, and device code, which is execute in parallel on the GPU. The device code consists of C functions, called kernels. On the new GPU architectures, the kernels can be executed concurrently. During program execution, the kernels are called from host code. The size of the grid and the number of threads per block can be defined per kernel at runtime.

CUDA allows to define host and device functions by extension of the function head. The possible function extensions are:

___global___: Called from host code and define a kernel.

___device___: Called from device code to define a device function.

___host___: Called from host code to define a host function.

28 1.5 GB or 3.0 GB global memory on the GTX 580 GPU.
29 64 KB constant memory on the GTX 580 GPU.
30 48 KB per SM on the GTX 580 GPU.
31 32768 32 bit register per SM on the GTX 580 GPU.

___host___ ___device___: Called from host or device code to define a host and device function.

Variables can be one, two, or three-dimensional. Possible extensions for variables are:

no extension: The variable is stored in register or local memory.

___shared___: The variable is stored in shared memory.

___constant___: The variable is stored in constant memory.

___device___: The variable is stored in device memory.

The function *___syncthreads()* can be used for thread synchronisation in a block and is required for shared memory access. For global synchronisation of the threads, new kernels need to be defined. To prevent race conditions, atomic operations are available. For data transfer from host memory to device global memory, copy functions are available: host to device, device to host and device to device. For efficient data transfer between CUDA and OpenGL vertex buffer objects (VBOs) are used.

2.4.4 Code Optimization

In section 2.4.2 different memory spaces were discussed. The optimising the usage of memory spaces on the grid, block and thread layer and optimising of the memory accesses can improve the performance of CUDA program. Additionally, memory allocation on the device and data copy from host to device (or device to host) is very expensive. Therefore, if possible, memory transfer should be avoided or asynchronous and overlapping transfers should be uses to improve the performance. Moreover, coalesced access to global memory (see [NVI11a]) should be used to reduce the memory transfer cost on the thread layer.

The instructions for flow control (e.g. if, for, switch and while) can cause the threads of the same warp to branch. In this case, the threads of the same warp with different execution paths need to be executed sequentially. This increases the number of required instructions and reduces the performance.

CHAPTER 3

Mesh Simplification

Highly detailed geometric models are very popular in interactive applications such as computer games or internet shops. These models are usually represented as triangle meshes. To render several of these models at real-time frame rates, level-of-detail (LOD) techniques are commonly used. For multiple smaller objects, static LODs are usually the method of choice. Each model is represented at several resolutions. During rendering, a level is chosen per object based e.g. on the distance to the viewer (see Figure 3.1). Simplification algorithms can be used to automatically generate the different resolutions, so that designers only need to model the finest level.

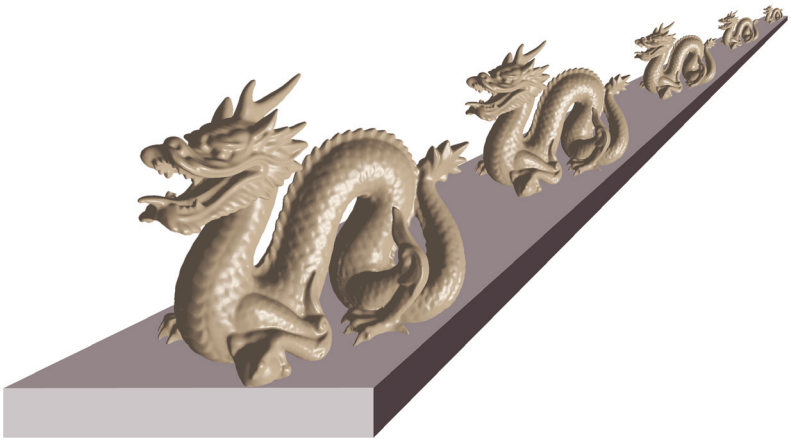

Figure 3.1: With increasing distance, coarser approximations of the model can be used.

The main goal of simplification algorithms is to reduce the number of rendered triangles while introducing little or no visual difference. The maximum screen-space deviation ϵ_s in pixels and the simplification error ϵ between simplified and original model induce a minimum view distance d. The distance also depends on the field of view α and the screen resolution r. It can be computed by $d = \epsilon \frac{r}{2\epsilon_s \tan \frac{\alpha}{2}}$.

As the distance depends on the maximum error, a simplification error that reduces the model up to a specified error bound is desirable in this context. An efficient way to

reduce the model to a specified ϵ are the quadric error metrics proposed by Garland and Heckbert [GH97a]. The distance between an simplified and original model is estimated by accumulating quadratic plane distances. While this is an overestimation of the real error, the specified error is still an upper bound. The simplification is based on successive edge collapse operations. After an edge of the model is collapsed, the degenerated faces are removed. This process is repeated, until no more collapses are possible without exceeding the error bound. Despite the existance of more accurate algorithms to estimate the simplification error, the quadric error metrics are still commonly used because of the possibility to compute an optimal placement of the collapsed vertex.

Due to the processing time, simplification algorithms are normally used as a preprocessing step. Considering that the average performance of the quadric error metrics simplifier is about 50,000 operations per second, the LOD generation needs 20 seconds for a model with one million vertices. Therefore, the levels are stored on disk and loaded at program startup. As the LODs in total normally contain as many triangles as the original mesh, the loading times are doubled. While this is unproblematic when loading from a local disk, it might be inacceptible for online applications. Here the LODs need to be generated from the transferred original mesh. Although almost every customer level computer contains a graphics card that can be used for general purpose computations, edge-collapse simplification algorithms are still working sequentially. This is mainly due to the fact that a significant amount of neighborhood information is required to compute an optimal ordering of operations.

The main contribution is a high-quality parallel simplification algorithm using edge collapse operations. Based on the observation, that the ordering only needs to be preserved locally, all possible edges are determine and collapse in parallel. The collapsed vertex locations and the simplification errors are computed using the quadric error metric. This leads to an exceedingly fast high-quality simplification algorithm. Using the proposed implementation, a complete set of 10 detail levels from the welsh dragon model (2.2 million faces) can be genereted within 0.73 seconds. Figure 3.2 shows a subset of the generated levels.

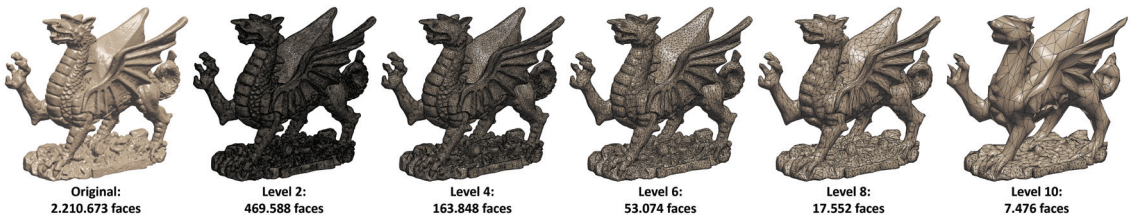

Figure 3.2: A subset of the 10 detail levels for the Welsh Dragon generated with the proposed algorithm.

3.1 Previous Work

Mesh simplification is one of the fundamental techniques for real-time rendering of polygonal models. There is an extensive amount of methods that mainly focus on accurate bounds of the simplification error. A detailed review of simplification algorithms is given by Luebke [Lue01]. As the focus of this thesis is real-time simplification, only the methods that would be suitable candidates are brief discuss.

3.1.1 Vertex Clustering

Rossignac and Borrel [RB93] proposed to use uniform vertex clustering. The bounding box of the model is subdivided using a regular grid and all vertices inside the same grid cell are collapsed to their mean. Low and Tan [LT97] proposed a weighted vertex clustering to preserve edge features that are not aligned with the grid. While uniform clustering is relatively fast and a precise upper bound for the simplification error can be given, a further reduction in flat regions would be possible. An adaptive vertex clustering using octrees was later proposed by Schaefer and Warren [SW03]. The runtime is even higher than using a BSP tree, but the quality of the simplified mesh can almost compete with edge collapse simplification. Additionally, DeCoro and Tatarchuk [DT07] proposed a parallel implementation of vertex clustering on the GPU. It is based on the octree clustering approach of Schaefer and Warren [SW03] by implementing an efficient octree data structure on the GPU. While the performance is extremely high, it still has the same quality problems as all vertex clustering algorithms:

- Although geometry can be optimized, the optimization of the topology is only limited, given by the cluster structure (e.g. grid or octree). This significantly increases the requiered number of faces.
- The cell size is defined by the factor of 2, this reduces the number of possible quality levels.
- No smooth transitions between LODs possible.

3.1.2 Quadric Error Metrics

Garland and Heckbert [GH97a] as well as Popović and Hoppe [PH97] introduced the vertex pair contraction. The contraction operation is combined with the introduced quadric error metric. It allows a flexible control over the geometric error and can be used to calculate optimal vertex positions. This approach has become the most common technique for the simplification of triangle meshes. Although the performance is lower, it overcome the quality problems of the vertex clustering. Later Garland and Heckbert extended their approach to an arbitrary number of vertex attributes [GH98]. While the generated

approximations are superior to vertex clustering at the same number of triangles, the simplification performance is significantly lower. On the other hand, the required levels can be generated using a single simplification sequence from the original model to the coarsest level.

Lindstrom [Lin00] proposed a combination of vertex clustering with error quadrics to improve the placement of the clustered vertices. Nevertheless, a high number of triangles is used in flat regions. Shaffer and Garland [SG01] proposed to overcome this problem by using a BSP tree instead of a uniform grid. The runtime is increased by a factor of three compared to uniform clustering, but the method is still faster than edge collapse simplification.

Garland and Shaffer developed a multiphase algorithm [GS02] which combines vertex clustering with a subsequent edge contraction to generate a drastic simplification. While this is faster than edge collapses alone, it can only be used to generate a single detail level.

3.2 Instant Level-of-Detail[32]

The simplification algorithm generates the simplified models by collapsing all non-conflict edges in parallel. Figure 3.3 shows the principle of an edge collapse operation col_v, which contracts an edge, connecting the vertices v and v_u, into a point. By applying col_v the adjacent faces f_l and f_r of the vertices v and v_u disappear and the position of the collapse vertex $\overline{v}$ is computed by minimizing the costs of the collapse operation col_v. To provide control over the simplification error and to evaluate the costs for col_v a suitable measure is required. Garland and Heckbert [GH97a] proposed a quadric error metric that estimates the distance between simplified and original mesh.

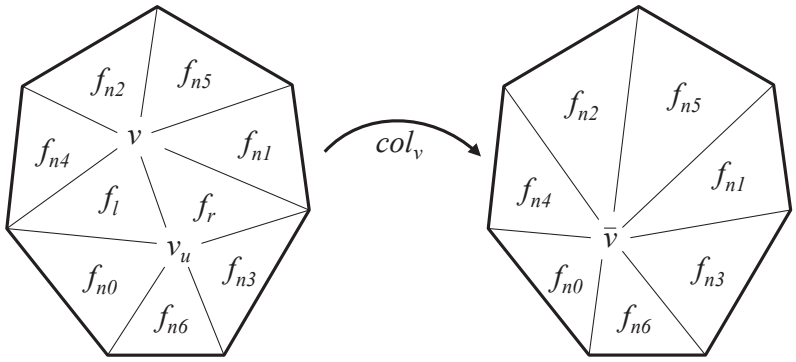

Figure 3.3: Edge collapse. The edge defined by vertex v and v_u is collapsed into the vertex $\overline{v}$. During the collapse operation col_v the position of $\overline{v}$ is estimated concerning a specified quadric error metric $\overline{Q}$.

32 In proceedings of Vision Modeling and Visualization (VMV2011) [GDG11].

3.2.1 Quadric Error Metric

The quadric error metric approximation is based on the distances of the simplified vertex to the planes defined by the adjacent triangles in the original mesh. Let $P(v)$ be the set of planes adjacent to mesh vertex v, then the maximum error can be estimated with the sum of squared distances:

$$\Delta(\mathbf{v}) = \Delta[v_x\ v_y\ v_z\ 1]^T = \sum_{\mathbf{p}\in P(v)} (\mathbf{p}^T\mathbf{v})^2, \tag{3.1}$$

where $\mathbf{p} = [a\ b\ c\ d]^T$ is the implicit plane equation $ax + by + cz + d = 0$ in normalized form, i.e. $a^2 + b^2 + c^2 = 1$. Note, that the coefficients a, b and c are the plane normal and d the signed distance between the origin and the plane. The sum of the squared distances can be transformed into a quadratic form:

$$\begin{aligned}
\Delta(\mathbf{v}) &= \sum_{\mathbf{p}\in P(v)} (\mathbf{v}^T\ \mathbf{p})(\mathbf{p}^T\ \mathbf{v}) && (3.2)\\
&= \sum_{\mathbf{p}\in P(v)} \mathbf{v}^T\ (\mathbf{p}\mathbf{p}^T)\ \mathbf{v} && (3.3)\\
&= \mathbf{v}^T \left(\sum_{\mathbf{p}\in P(v)} \mathbf{Q}_p \right) \mathbf{v}, && (3.4)
\end{aligned}$$

where $\mathbf{Q}_p$ is the convariance matrix of the planes $\mathbf{p}$ in $P(v)$:

$$\mathbf{Q}_p = \mathbf{p}\mathbf{p}^T = \begin{bmatrix} a^2 & ab & ac & ad \\ ab & b^2 & bc & bd \\ ac & bc & c^2 & cd \\ ad & bd & cd & d^2 \end{bmatrix} \tag{3.5}$$

By summing up the error quadric $\mathbf{Q}_p$ for a set of planes $P(v)$ associated to a vertex v it is possible to represent the corresponding error quadric by a single matrix $\mathbf{Q}_v$.

To prevent a degeneration of the mesh boundary, we use the same approach as Garland and Heckbert. In addition to the vertex quadrics, a boundary quadric is calculated for each boundary edge. It is computed from a virtual plane that is orthogonal to the triangle plane. Let v_1 and v_2 be the vertices of the boundary edge and v_3 the third vertex of the only adjacent face. Then the normal equation of the virtual plane is:

$$t_x x + t_y y + t_z z - n \cdot v_1 = 0, \tag{3.6}$$

with

$$e_1 = \frac{v_2 - v_1}{\|v_2 - v_1\|}, e_2 = \frac{v_3 - v_1}{\|v_3 - v_1\|}, t = \frac{e_2 - (e_1 \cdot e_2)e_1}{\|e_2 - (e_1 \cdot e_2)e_1\|}. \tag{3.7}$$

The quadric error metrics can be generalized to arbitrary dimensions [GH98]. The general quadric $\mathbf{Q}_p$ can be written as:

$$\mathbf{Q}_p = \begin{pmatrix} \mathbf{A} & \mathbf{b} \\ \mathbf{b}^T & d \end{pmatrix}, \tag{3.8}$$

with

$$\mathbf{A} = \mathbf{Id} - e_1 e_1^T - t t^T \tag{3.9}$$

$$\mathbf{b} = (v_1 \cdot e_1) e_1 + (v_1 \cdot t) t - v_1 \tag{3.10}$$

$$d = v_1 \cdot v_1 - v_1 \cdot e_1 - v_1 \cdot t \tag{3.11}$$

In order to compute the cost of collapsing a pair of vertices v and v_u, the associated error from the vertex quadrics $\mathbf{Q}_v$ and $\mathbf{Q}_u$ can be derived. The total sum of squared distances is $\overline{\mathbf{Q}} = \mathbf{Q}_v + \mathbf{Q}_u$. In addition, $\overline{\mathbf{Q}}$ can be used to find the optimal position of the collapsed vertex $\overline{\mathbf{v}}$. The optimal vertex minimizes the sum of squared distances $\Delta(\overline{v})$. This translates into solving the following linear equation system:

$$\nabla \overline{\mathbf{Q}} \ (\overline{v}) = \begin{pmatrix} \mathbf{A} & \mathbf{b} \\ 0 \cdots 0 & 1 \end{pmatrix} \begin{pmatrix} \overline{v} \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \end{pmatrix} \tag{3.12}$$

3.2.2 Overview

Traditionally, edge collapse simplification works by sequentially collapsing the edge with the minimum induced error. When generating a specific level-of-detail, the simplification stops when collapsing the next edge would exceed the error threshold. This can also be seen as successively collapsing the vertex v_a that is adjacent to the edge with the minimum error with the other vertex of that edge v_b. With this, the minimal adjacent edge error can be associated with the vertex. In the example, v_a and v_b will have the same error which is also the global minimum. The quadric error of the collapsed vertex is then the minimum of the other edges adjacent to the collapsed one. By construction, the vertex quadrics are monotonically increasing and thus the quadric error of the adjacent edges will be at least as high as that of the collapsed edge. This means that the edge with the minimal error can not only be collapsed, but also all other edges that have a local minimum of the error below the given threshold. Due to the monotonically increasing error of the other edges, these will also be collapsed in the sequential algorithm. The overall algorithm therefore works as follows:

1. Compute vertex quadrics.

2. Compute placement and error for each edge.

3. Find local error minima below threshold and collapse edges.

4. Continue with 2. until no collapses can be performed.

To perform the quadric computations and the edge collapses in parallel, an adequate data structure is require. In addition to vertices and faces, a data structure for the edges of the mesh is needed. To compute the target placement for an edge collapse, access to the vertices and the associated vertex quadrics from the edge is required. Theoretically, all edges from a vertex need to be accessed in order to find the edge with minimal error. Since this is computed for all vertices, the atomic operations can be used and process all edges instead. The same holds for removing the degenerated faces. There the indices can be simply updated and the faces removed using a compaction algorithm.

3.2.2.1 Connectivity Data Structure

As discussed above, the edges of the mesh need to be extract. In addition, the boundary edges for the boundary quadrics need to be determined. Initially, an indexed face set (IFS) is loaded and transfered it to the GPU. This means that it are attributes per vertex and three indices per triangle available. Algorithm 1 shows how the edge information is build from this data.

```
foreach face f in parallel do
    i1,i2,i3 = get_face_indices(f)
    edge1 = create_edge(min(i1,i2),max(i1,i2),i3)
    edge2 = create_edge(min(i2,i3),max(i2,i3),i1)
    edge3 = create_edge(min(i3,i1),max(i3,i1),i2)
RADIX SORT edges in parallel by imax
RADIX SORT edges in parallel by imin
foreach edge e in parallel do
    ep = get_previous_edge(e)
    if ep==e
        set_edge_flag(e, 0)
    else
        set_edge_flag(e, 1)
        if get_previous_edge(e)==e
            set_single_flag(e, 0)
        else
            set_single_flag(e, 1)
COMPACT edges in parallel
```

Algorithm 1: Parallel generation of the edge data structure.

First an edge for each of the three half-edges of a triangle is generated. Then the edges are sorted by their higher vertex i_{max} and then by the lower one i_{min} using radix sort. Since radix sort is stable, the edges are now sortened in lexicographic order and duplicates

can be marked for removal. During this process, the third vertex of the face is stored and single edges are flaged.

The complete data structure required for the edge setup and the allocated memory (per entry) are shown in Table 3.1. The input data are the *vertex* and the *index buffer*. Both are stored as vertex buffer objects (VBOs) and are therefore separated from all other data. All generated data are stored per edge, where the opposite vertex and single edge flag are only required to compute the vertex quadric. The total memory required during this phase is $246 + 4k$ bytes per vertex, where k is the number of vertex attributes, or 222 bytes in addition to the IFS. After removing the duplicate edges and freeing the temporary arrays, this is reduced to $63 + 4k$ bytes per vertex, or 39 additional bytes.

buffers	**elements**	**bytes per entry**
vertices	vertex VBO	$4k$
faces	index VBO	12
edges	vertex index ($\times 2$)	8
	vertex index (opposite)	4
	single edge flag	1
temporary per edge	sorting edge ($\times 3$)	12
	sort order	4
	sort key	4
	sort prefix sum (scan)	4

Table 3.1: Mesh data structure after generating the edge information, where k is the number of vertex attributes.

3.2.3 Parallel Simplification

The algorithm is subdivided into several consecutive steps to implement the simplification on massively parallel hardware. The partitioning is required for thread synchronisation while each step can be processed completely in parallel. Figure 3.4 shows the steps of the algorithm. The first step is to compute the vertex quadrics. These are sums of all adjacent face and boundary edge quadrics. After this step, the opposite vertex and single edge flag arrays can be deallocated. Then the parallel simplification loop starts. First, the optimal

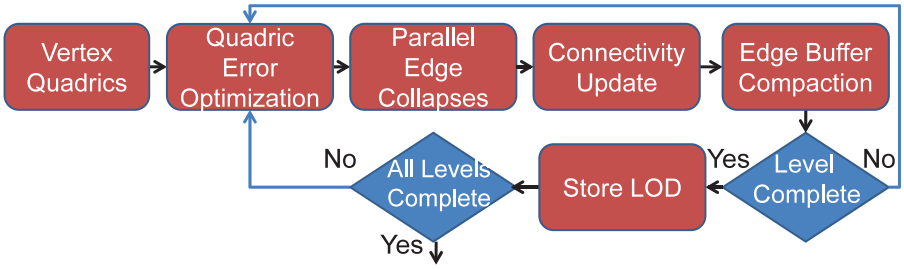

Figure 3.4: The steps of the algorithm.

collapse position and cost are computed per edge. Then the local cost minima are found and the associated collapses are performed. After updating the face and edge connectivity, the collapsed edges are removed. If no further collapses are possible without exceeding the threshold, the current level can be copied to a vertex and index buffer. The simplification loop is then continued with an increased threshold, until all required LODs are generated.

The complete data structure maintained during simplification is shown in Table 3.2. For each vertex the quadric, the index of the adjacent edge with the minimal cost are required. Due to the symmetry of $\mathbf{Q}$ only the upper triangular matrix is stored. This only need $2k^2 + 6k + 4$ bytes instead of $4k^2 + 8k + 4$ bytes per quadric. Again, k is the number of vertex attributes. In addition, inactive vertices need to flagged to generate an IFS for each level-of-detail. To determine the minimal cost edge, the minimal cost over all adjacent edges are needed as temporary data. Later, when the collapse target of each vertex is required, this array can be reused. For the faces, only the flag for degenerated triangles during construction of the indexed face set is required. Finally, the collapse cost and the optimal placement $\overline{\mathbf{v}}$ for each edge are stored, together with the collapse state and a flag to mark degenerated edges. In total $2k^2 + 22k + 120$ bytes per vertex are required, which is 204 bytes for $k = 3$ and 324 bytes for $k = 6$.

buffers	**elements**	**per entry (bytes)**
vertices	vertex VBO	$4k$
	vertex quadric	$2k^2 + 6k + 4$
	min edge ID	4
	active flag	4
	min edge cost / target vertex	4
faces	index VBO	12
	active flag	4
edges	vertex indices	8
	edge cost	4
	optimal placement	$4k$
	collapse state	4
	active flag	4
temporary	perfix sum (scan)	4

Table 3.2: Data structure used during simplification loop.

3.2.3.1 Vertex Quadrics

The first step is the computation of the vertex quadrics which are the sum of all adjacent face quadrics. In addition, the boundary quadrics need to be computed and accumulated. As the boundary edges are already flagged, simply the virtual plane need to be computed and add the corresponding quadric to the edge vertices. The complete parallel vertex quadric calcuation is shown in Algorithm 2. Note, that atomic float addition is used in the implementation, parallel binning could also be used to accumulate the quadrics for

GPU this operation.

```
foreach face f in parallel do
    i1,i2,i3 = get_face_indices(f)
    quad = compute_face_quadric(f)
    add_quadric(i1, quad)
    add_quadric(i2, quad)
    add_quadric(i3, quad)
foreach single_edge e in parallel do
    i3 = get_opposite_vertex(e)
    quad = compute_boundary_quadric(imin[e], imax[e], i3)
    add_quadric(imin[e], quad)
    add_quadric(imax[e], quad)
```

Algorithm 2: Parallel calculation of the vertex quadrics.

3.2.3.2 Quadric Error Optimization

The first step of the simplification loop is calculating the edge cost and to determine the possible collapses. The edge e can be collapse if its *cost* is at most ε^2, where ε is the error threshold. If the cost is below the threshold, the edge is marked as a collapse candidate. To compute the cost, first the optimal placement $\overline{\mathbf{v}}$ need to found and then evaluate the quadric $\overline{\mathbf{Q}}$ for $\overline{\mathbf{v}}$. The edge quadric $\overline{\mathbf{v}}$ is the sum of the two vertex quadrics $\mathbf{Q}_{v_{min}}$ and $\mathbf{Q}_{v_{max}}$. Since $\mathbf{A}$ is a symmetric, positiv-semidefinite matrix, the LDL decomposition (cholesky decomposition without square roots) is used to solve the linear equations. The symmetry of $\overline{\mathbf{Q}}$ is also exploited when calculating the quadric error. Then for each vertex the minimal edge cost is stored using the atomic min operation. Similar to the vertex quadric accumulation, binning on GPU is used without atomic operations. Algorithm 3 shows the parallel quadric error minimization.

```
foreach edge e in parallel do
    quad = calc_edge_quadric(vertex_quadric[vmin],vertex_quadric[vmax])
    collapse_pos[e] = optimize_pos(quad)
    edge_cost[e] = calc_cost(quad, collapse_pos[e])
    if edge_cost[e] ≤ ε²
        collapse_state[e] = collapse
        atomic_min(min_edge_cost[vmin], edge_cost[e])
        atomic_min(min_edge_cost[vmax], edge_cost[e])
    else
        collapse_state[e] = no_operation
```

Algorithm 3: Parallel quadric error minimization algorithm.

3.2.3.3 Parallel Edge Collapses

The collapse of an edge is only possible if its cost is a local minimum. As the minimal cost per vertex is already stored, now the edge with the associated cost need to be determined.

Then the edge can be collapsed if both vertices store a reference to it as minimal cost edge. For all other edges, the collapse flag is cleared. After the possible collapses are determined, they can be applied to the mesh. The collapse operation then simply moves vertex $v = v_{min}$ to its new position $\overline{\mathbf{v}}$, which is stored with the edge and marks $v_u = v_{max}$ as invalid and stores v as its target index. The new quadric $\mathbf{Q}_{\overline{v}}$ is the sum of the vertex quadrics $\mathbf{Q}_{v_{min}}$ and $\mathbf{Q}_{v_{max}}$. Algorithm 4 shows the parallel processing of the edge collapse operations.

```
foreach edge e in parallel do
   if collapse_state[e] == collapse
      cost = get_edge_cost(e)
      if min_edge_cost[v_min] == cost: edge_ID[v_min] = e
      if min_edge_cost[v_max] == cost: edge_ID[v_max] = e
foreach edge e in parallel do
   if collapse_state[e] == collapse
      if edge_ID[v_min] == e and edge_ID[v_max] == e
         v̄ = collapse_pos[e]
         vertex_quadric[v_min] += vertex_quadric[v_max]
         collapse_target[v_max]=v_min
         vertex_active[v_max]=false
```

Algorithm 4: Parallel edge collapse algorithm.

3.2.3.4 Connectivity Update

After performing the collapses, the indices of the adjacent faces and edges need to be updated (i.e. $f_{n1} - f_{n6}, f_l$, and f_r in Figure 3.3). Algorithm 5 shows the parallel index update. Here the collapse targets set is used during the edge collapses. If a face or edge becomes degenerated it is marked as invalid and will be removed in a subsequent stage.

```
foreach face f  in parallel do
   update_indices(f,collapse_target)
   if degenerate(f): face_vaid[f]=false
foreach edge e  in parallel do
   update_indices(e,collapse_target)
   if degenerate(e): edge_vaild[e]=false
```

Algorithm 5: Parallel index update.

3.2.3.5 Edge Buffer Compaction

The final step of the adaption is the compaction of the edge buffer. The removal of invalid edges is not only necessary for performance reasons, but also tells us when the simplification has finished. Algorithm 6 shows the edge compaction. At the end the storage for the old edge buffer can be freed and thus gradually reduce the memory consumption. If no duplicate or degenerated edge was found, the compaction is skipped and the LOD

```
RADIX SORT edges in parallel by i_max
RADIX SORT edges in parallel by i_min
foreach edge e in parallel do
   e_p = get_previous_edge(e)
   if e_p==e or degenerate(e)
      set_edge_flag(e, 0)
   else
      set_edge_flag(e, 1)
COMPACT edges in parallel
```

Algorithm 6: Edge compaction algorithm.

creation can be started. Otherwise, the simplification loop is continue. Note, that the edges are sorted after each fifth iteration only as the speedup from the removed duplicates is less than the time required for sorting.

A specialized in-place compaction algorithm (Section 5.2.2.4) is used, since the ordering does not need to be preserved. The main advantage besides a minor speedup is that these buffer need not to be duplicate.

3.2.3.6 LOD Creation

If no collapses were performed, the generated level can be stored. To store the mesh, first the vertex buffer is compacted according to the active flag of the vertices. The compacted vertices are directly stored in a vertex VBO. Then the indices are compacted according to the active flag of the faces and store them in an index buffer. If the number of faces is above a user specified threshold, the error threshold is doubled and another level is generated. Otherwise, all data structures can be freed except the original and generated VBOs and can now render them as static LODs.

3.2.4 Results

The test system consists of a 3.333 GHz Intel Core i7-980X CPU with 6 GB DDR3-1333 main memory and an NVIDIA GTX580 (841/4204MHz). CUDA is used to implement the parallel simplification and generate indexed face sets for OpenGL. For comparison with loading precomputed LODs, a SATAII hard disk (8.5ms/64MB/7200rpm) is used with approximately 100 MB/s read speed. Table 3.3 gives an overview of the simplified models. All models use position and normal as vertex attributes ($k = 6$). Additionally, the original model size (IFS) and the size of the generated LODs are shown.

Figure 3.1, 3.2, and 3.7 show the generated LODs and Table 3.4 gives an overview of all generated levels with their number of faces. The first level was generated with an error threshold of $\varepsilon = 0.1\%$ of the bounding box diagonal. With each level, the threshold is doubled and LOD generation is stopped as soon as a level contains less than 10k triangles. Depending on the complexity of the model, 8 to 10 levels are generated this way.

model	# vertices	# faces	IFS	LODs
Apache	445,836	807,365	19.4 MB	16.7 MB
St. Dragon	437,645	871,414	19.9 MB	17.5 MB
Buddha	543,652	1,087,716	24.9 MB	24.7 MB
Welsh Dragon	1,105,352	2,210,673	50.5 MB	45.9 MB
Youthful	1,728,305	3,411,563	78.6 MB	70.6 MB
Awakening	2,057,930	4,060,497	93.5 MB	81.4 MB

Table 3.3: Models used for evaluation.

level	Apache	St. Dragon	Buddha	Welsh Dragon	Youthful	Awakening
original	807,365	871,414	1,087,716	2,210,673	3,411,563	4,060,497
level 1	321,072	328,733	454,844	871,236	1,255,238	1,514,414
level 2	187,542	190,002	273,860	469,588	750.257	893,595
level 3	112,622	110,765	162,282	278,126	455,584	517,640
level 4	66,433	63,105	92,352	163,848	267.815	284,144
level 5	39,359	35,218	51,124	95,372	152,008	149,370
level 6	22,701	19,096	27,504	53,074	83,185	75,750
level 7	12,908	10,058	14,508	30,768	42,530	37,273
level 8	7,249	5,132	7,608	17,552	19,368	15,203
level 9	-	-	-	11,500	8,982	5,639
level 10	-	-	-	7,476	-	-

Table 3.4: Generated levels with number of faces.

Table 3.5 shows a comparison of the proposed approach to the reference QSlim implementation of Garland and Heckbert [GH97a]. The runtime complexity of their approach is $\mathcal{O}(N \log N)$, due to the required priority queue. In contrast to that, the complexity of the proposed algorithm is $\mathcal{O}(N)$ (Figure 3.6) as it only use radix sorting with fixed key length. Compared to CPU simplification, it achieve a speedup of 30 to 40 and can perform up to 2 million collapses per second. The simplification time is similar to the transfer time of the levels from HDD to the GPU and is significantly faster than the transfer time over a network. The total amount of consumed graphics memory is approximately 7 to 8 times

	QSlim		proposed approach			
model	time (s)	k Op/s	memory	time (s)	k Op/s	speedup
Apache	8.0	55.7	136.4 MB	0.29	1537	28
St. Dragon	8.0	54.7	139.9 MB	0.28	1564	29
Buddha	10.0	54.4	174.3 MB	0.35	1552	29
Welsh Dragon	22.2	49.8	354.2 MB	0.73	1519	31
Youthful	35.8	48.3	550.7 MB	0.89	1941	40
Awakening	43.9	46.9	655.6 MB	1.03	2003	43

Table 3.5: Comparison of processing time and the number of operations per second with QSlim tested of the test system.

higher than that of the original models. This is equal slightly less than the main memory consumed by CPU quadric error metrics.

Compared to the vertex clustering algorithm of Lindstrom [Lin00] the speedup is approximate 10. Consequently, the proposed method is 20 times faster than using BSP trees [GS02] and 70 times faster than octree vertex clustering [SW03]. Compared to the GPU implementation of vertex clustering by DeCoro and Tatarchuk [DT07], the proposed method is 3 to 4 times slower. They implemented the method however for vertex position only ($k = 3$). With increasing quadric dimension, the difference vanishes since most of the time is spent to optimize the vertex placement. In addition, even octree vertex clustering requires slightly more triangles to achieve the same quality.

Finally, the runtime of each step is analyzed of the adaption and rendering algorithm in Figure 3.5. Already for $k = 6$, the most time consuming part of the algorithm is the quadric error minimization. As the LDL decomposition has a time complexity of $\mathcal{O}(N^k)$, this dominates the runtime for higher number of attributes.

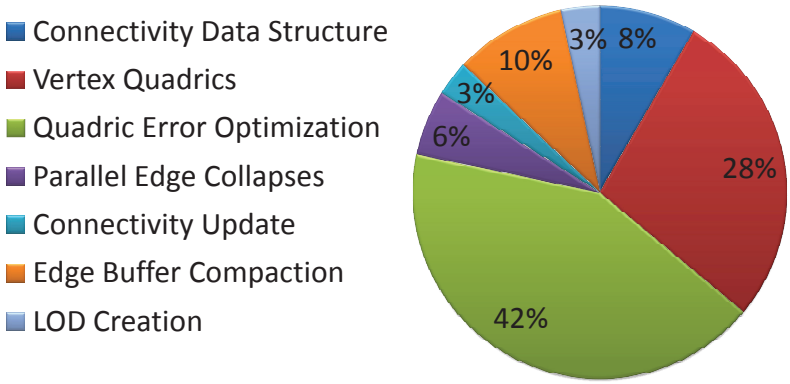

Figure 3.5: Relative time of the adaption steps compared to rendering.

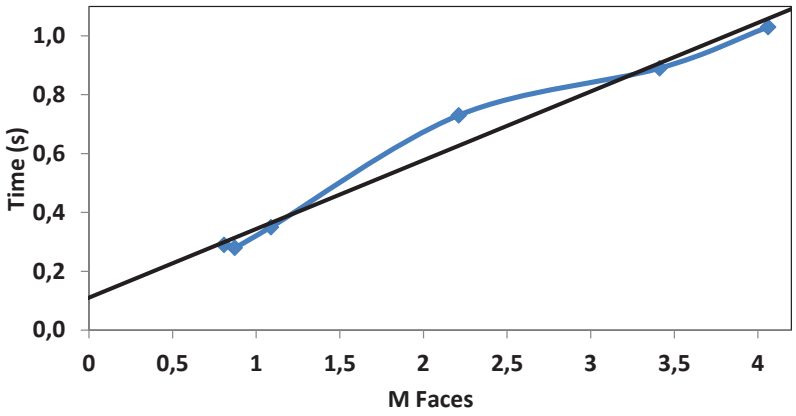

Figure 3.6: Processing time and number of faces of the proposed algorithm for the models from Table 3.5.

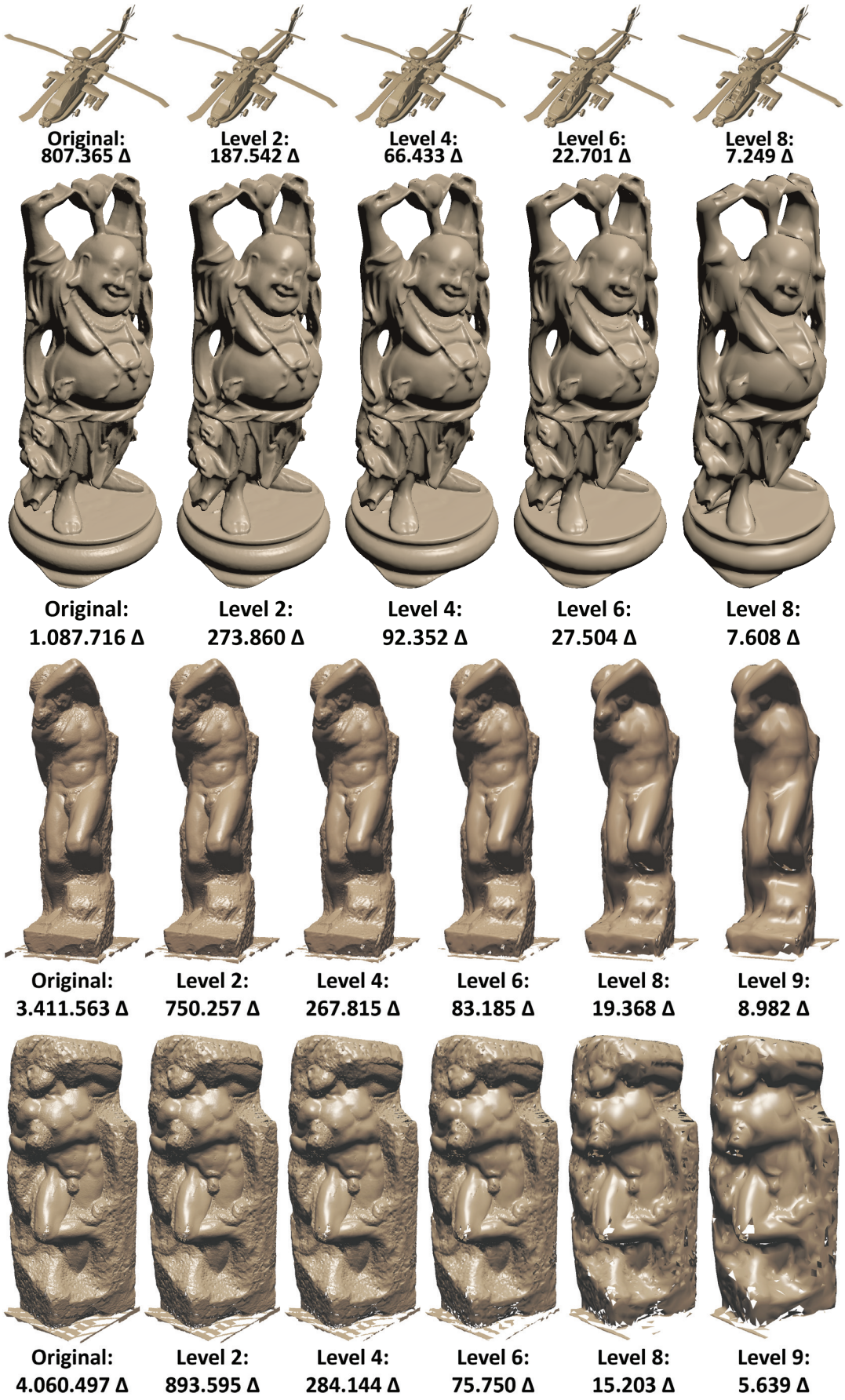

Figure 3.7: Each second generated LOD of the Apache, Buddha, Youthful and Awakening model. The first level is the original model.

3.2.5 Conclusion and Limitations

In this chapter a parallel implementation of the quadric error simplification developed by Garland and Heckbert [GH97a] was proposed. By collapsing all edges with a local minimum of the collapse cost, the generated meshes are identical to those produced by the sequential algorithm for a given error bound. On a customer level graphics card, the method can generate a set of LODs for a model containing over 4 million faces in less than a second. This is comparable to loading the generated LODs from disk and significantly faster than network transfer.

The main limitation of the proposed algorithm is that the computation of the target placement is rather expensive. With increasing number of attributes, this dominates the total runtime. Another limitation is, that the triangle flips are not checked during simplification. While this was unproblematic for the models I examined, it might produce visible artifacts for others.

A possible extension of the proposed method would be the addition of vertex pair contractions. These could be integrated by adding an additional set of virtual edges before simplifying the mesh for a level. The maximum vertex distance would then be in the range of the error threshold.

CHAPTER 4

Iso-Surface Extraction and Simplification

During the last decades, the resolution of volume data sets has constantly grown which also increased the memory requirements and computation times. A high number of triangles is generated from such data sets when using iso-surface extraction algorithms. Such algorithms are based on selecting an iso-value for the density and generating a triangle mesh that approximates the corresponding iso-surface. Often finding the correct iso-values requires user feedback to interactively determine the optimal one. In such cases, fast and efficient algorithms are required to reduce feedback time.

For high quality meshes, the classical marching cubes algorithm is often used. It is based on a linear interpolation of the density between the vertices of the volume. Unfortunately, it produces a mesh than contains a high number of sliver triangles. This also results in a high number of faces, which increases rendering time and memory consumption of the mesh. While other approaches, especially the dual marching cubes algorithm, reduce the number of triangles and improve their shape, the approximation error is higher. Instead of improving the mesh during extraction, another approach is to reduce the number of triangles by collapsing sliver and co-planar ones after iso-surface extraction. The classical approach is to use a mesh simplification algorithm after the whole iso-surface is extracted. The simplification step itself generates a finial mesh, that is a further approximation of the initial iso-surface. One of the main problems is the huge memory consumption when starting to simplify a detailed iso-surface mesh. This can only be solved by using out-of-core simplification algorithms, especially stream simplification methods.

Another possibility is to directly simplify the mesh while it is being reconstructed. The volume is either processed using a plane sweep algorithm or by partitioning it into a regular grid or octree. As the processing front of the iso-surface extraction algorithm needs to be fixed during simplification, the vertices on the current boundary must not be collapsed. This however leads to strong artifacts because the neighborhood of the vertices was already strongly simplified. The main effect is that the number of vertices along the boundaries remains high and a lot of sliver triangles appear in that regions. The so-called tandem algorithm alleviates this problem by introducing a time lag. The simplification

error is reduced for vertices close to the current front and thus the vertex density gradually increases. When the front proceeds, the vertices on the previous boundary can now be better simplified as their neighbors were not reduced too much. While the mesh quality is better, the error falloff constitutes a trade off between memory consumption and processing time on one side and mesh quality on the other.

Instead of using a time lag, the proposed algorithm locally block only those collapse operations that might not be performed when simplifying the complete mesh. Traditional simplification algorithms perform the collapse operations sorted by their introduced error. As the operations are local, the same mesh can be produced by any global operation ordering, as long as the local order remains. The proposed algorithm exploit this property and simply block all operations in the direct neighborhood of the processing front. When additionally enforcing the correct local order, the simplification error automatically decreases for vertices close to the current boundary (see Figure 4.1). This way, the result will be identical to a simplification of the complete mesh and the operations are performed as soon as possible.

Figure 4.1: The interleaved iso-surface extraction with the locally blocking stream simplification of the proposed algorithm. Notice the automatic increase of the triangle density towards the processing front of the extraction.

Recent iso-surface extraction and simplification algorithms were implemented on the GPU. In this chapter, a massively parallel algorithm is proposed, which combined iso-surface extraction and simplification. The main contributions of the proposed approach are:

- A high quality out-of-core iso-surface extraction algorithm running within seconds on current GPUs.
- A parallel stream simplification algorithm to directly reduce the number of triangles that produces the same result as a simplification of the whole mesh.

4.1 Previous Work

As mentioned above, the proposed approach combines techniques from iso-surface extraction and simplification algorithms. In this section a short overview of recent approaches in both fields is given.

4.1.1 Iso-Surface Extraction

The marching cubes algorithm [LC87] was designed for volume data sets defined on a rectilinear grid. The grid is a three-dimensional Cartesian discretized scalar field, often called voxel grid. The algorithm divides the whole data into cubes and processes them sequentially. In every cube, the scalar values are classified using an iso-value (see Section 2.2.1). One drawback of the classical marching cubes algorithm is the possibility of holes in the mesh. These are caused by ambiguous cases, when two or three vertices are above or below the surface but not connected by cube edges. Soon extensions were proposed to solve the ambiguities and generate hole free meshes [HGB93, MSS94, Pau94]. Another possibility is to use tetrahedrons [TPG99] instead of cubes, but this drastically increases the number of triangles. Later, Chernyaev also defined a topologically correct iso-surface based on tri-linear interpolation [Che95].

In addition, several methods were proposed to improve the performance. Wilhelms and van Gelder [WG90] introduced a modified branch-on-need-octree with min-max decisions to save calculation time. The methods of Shen et al. [SHLJ96] and Livnat [Liv99] further improved this idea. They use span spaces, by which the eight corners of a cube are reduced to a two-dimensional point. Then partitions (lattice elements) and k-d-trees allow a fast extraction of the iso-surface. To improve the mesh quality, Schaefer and Warren [SW04] proposed the Dual Marching Cubes algorithm. A dual grid lying in the primal grid and the use of the quadric error function combined with the method of Lindstrom [Lin00] for positioning dual vertices generate a mesh with better quality and less triangles. The algorithm is also very good in reconstructing sharp features (e.g. edges or corners) but the surface does not accurately approximate the tri-linearly interpolated iso-value. Mueller and Stark [MS91] proposed an adaptive method, which is called Splitting Boxes. Here the box (i.e. the cube) will be split and checked for just one sign-change in every edge. The splitting is applied recursively up to a given lowest level.

Recent approaches exploit the processing power of massively parallel graphics processors (GPUs). Reck et al. [RDG*04] proposed an algorithm to extract iso-surfaces from tetrahedral volumes. This method pre-selects the intersections of surface and voxel grid on the CPU and generates the mesh on the GPU using an interval tree. For rectilinear grids, a tetrahedalization is required that leads to a higher number of triangles and also introduces some artifacts. Johansson et al. [JC06] also use span spaces for pre-selection and pre-classification. The GPU is used for interpolation and their approach is not restricted

to tetrahedral grids anymore. Tatarchuk et al. [TSD07] propose a hybrid of marching cubes and marching tetrahedra running on the GPU. First they pre-process the scalar values to calculate the gradients. After voxelization, the cubes are used as input and tessellated into tetrahedra. The iso-surface is generated on GPU but again contains a higher number of triangles than the original marching cubes algorithm.

4.1.2 Simplification

Mesh simplification is one of the fundamental techniques for real-time rendering of complex polygonal models. There is an extensive amount of methods that mainly focus on accurate bounds of the simplification error. A review of simplification algorithms is given in a section 3.1.

4.1.3 Hybrid Algorithms

Current iso-surface extraction algorithms are based on a combination of the extraction itself with a simplification algorithm. The methods of Attali et al. [ACSE05] and Dupuy et al. [DJG*10] directly simplify the mesh while it is being generated. Attali et al. [ACSE05] uses a serial marching cubes, which runs in a tandem with a simplification algorithm. There is an extraction step and a simplification step alternating layer-by-layer. Their main contribution is to introduce a time-lag to the simplification. This means that edge collapses are delayed until the extraction front is further away from the cut plane and therefore results in a better approximation of the initial mesh. Dupuy et al. [DJG*10] this idea by using a load-balanced-cluster (Load Sharing Facility High Performance Computing) to parallelize extraction and simplification. In addition, they do not use a plane sweep algorithm but partition the volume using an octree. Another extension is that the parts of the mesh that cannot be simplified further are stored on disk to reduce the memory consumption.

4.2 Parallel Out-of-Core Iso-Surface Extraction and Simplification[33]

The core idea of the proposed approach is to interleave a massively parallel marching cubes algorithm with a massively parallel stream simplifier. As both algorithms run on the GPU, the proposed algorithm also minimize device to host communication since only the reduced mesh is transferred. Due to its simplicity, or partition strategy is a plane sweep algorithm, i.e. the data are processed in layers, although other partitionings would easily be possible.

33 Unpublished [UDG12].

4.2.1 Overview

The first module of the proposed method is the surface extraction itself that is based on a modified marching cubes algorithm. Implementing the algorithm in CUDA allows to extract the mesh of each cube in parallel. The proposed algorithm do not use textures but plain CUDA arrays which are easier to handle, especially during memory management. In addition, the algorithm support non-power of two rectangular volumes of arbitrary dimensions. Section 4.2.2 describes the implementation in detail.

The second module is the mesh simplification that receives the output of the first one as input. The simplification is based on edge collapse operations col_v that contract edges by collapsing two connected vertices v and v_u (see Figure 4.2). By applying col_v the adjacent faces f_l and f_r of the vertices v and v_u disappear. The position and normal of the collapse vertex $\overline{v}$ are computed by minimizing the quadric error metric [GH97a], which is also used to compute the collapse cost. An in depth discussion of the simplification algorithm is given in Section 4.2.3.

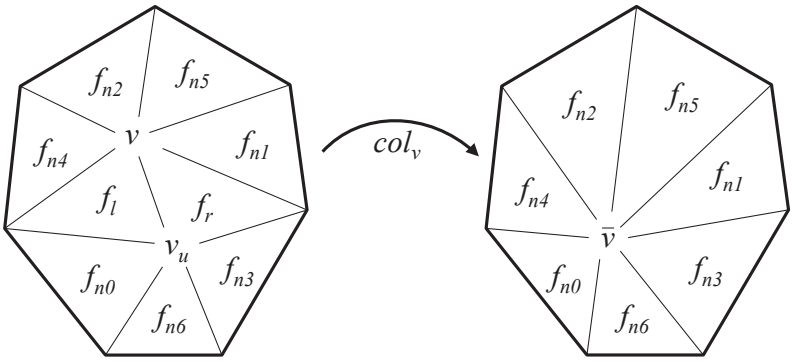

Figure 4.2: Edge collapse. The edge defined by vertex v and v_u is collapsed into the vertex $\overline{v}$ (comp. Figure 3.3).

4.2.2 Parallel Marching Cubes

The proposed algorithm divide the volume of dimension $dimX \times dimY \times dimZ$ into *layers* and *slices* like Attali et al. [ACSE05]. The *k-th slice* contains all vertices with the same y-coordinate. The *k-th layer* as a set of all vertices, edges and patches between or on the *k-th* and the *(k+1)-th* layer. So the volume comprises the layers from 0 to $(dimY - 1)$. The algorithm then process the layers in ascending order.

During surface extraction the algorithm group the slices and layers into partitions. Every partition is composed of N slices and $N - 2$ layers respectively, which are processed in two loops and mainly by two kernels (see Algorithm 7). The first kernel calculates the cube codes and, if necessary, the intersection between iso-surface and cube edges. The second one creates the triangles and thus builds up the mesh contained in the layers.

Note that two additional slices are required to compute surface normals from the

```
P, REST = partition(N)
kernel_cubecode_init()
for j = 0 to P − 1 do
    kernel_cubecode(N − 2)
    kernel_generate(N − 2)
    call_simplification_module()
kernel_cubecode_rest(REST)
kernel_generate_rest(REST)
call_simplification_module()
```

Algorithm 7: Parallel Marching Cubes Module.

gradients. Consequently the partitions overlap by one slice in each direction. As the vertices of the first slice were already calculated in the previous partition, the algorithm do however only need one additional slice in each partition. This means that when a partition has N slices only $N-2$ layers can be used for extraction (see Algorithm 7). In addition, separate kernels for the first and last slices are needed.

For each cube, the cube code kernel is executed (see Algorithm 8) using a thread block dimension of 16×16. Assuming that the four corners and four edges of the cube's bottom are already processed, every single thread just processes vertex v_4 and the edges e_4, e_7 and e_8 (see Figure 4.3). An exception are threads on the 'right' and/or 'front' border of a grid, that also process the edges e_5, e_6, e_9, e_{10} and e_{11} and the vertices v_5, v_6 and v_7. When $f : \mathbf{R}^3 \to \mathbf{R}$ is the scalar field and $\rho \in \mathbf{R}$ is the iso-value, every density value will be classified by:

$$\forall v \in \mathbf{R} : f'(f(v)) = \begin{cases} 1, & f(v) \geq \rho \\ 0, & f(v) < \rho \end{cases}. \tag{4.1}$$

If f' is zero, then the vertex is *below* the surface and if f' is one, it is *inside* or *on* the surface. When encoding the eight corner vertices of a cube into an eight-bit value, it uniquely defines the topology of the intersecting surface. If e.g. the corner vertices v_1 and v_4 are inside, the *cube code* would be $f'(f(v_7, v_6, v_5, v_4, v_3, v_2, v_1, v_0)) = 00010010$.

Every thread also calculates the gradient of vertex v_4. Threads at the grid's border

```
kth − layer = i ∗ (N − 2)
for j = 0 to N − 2 do
   foreach cube ∈ kth − layer  in parallel do
      calculate_gradients()
      generate_cubecode()
      if 0 < cubecode < 255
         calculate_intersections()
      shift_cubecode()
   kth − layer = kth − layer + 1
```

Algorithm 8: Cube code kernel.

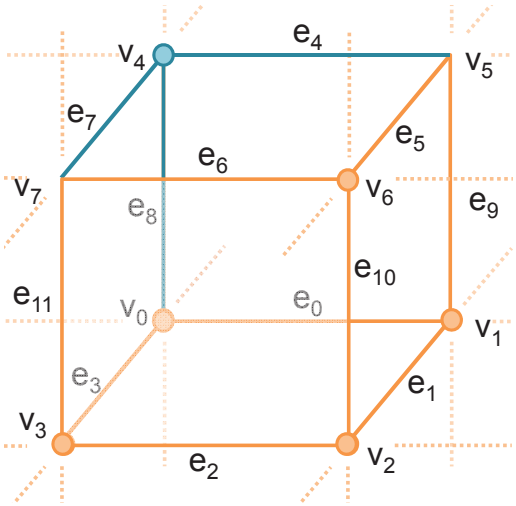

Figure 4.3: Edge and vertex indices similar to [Pau94]. Every single thread in a kernel just processes the blue corner and edges.

process up to four gradients. The gradients and the cube code always have to be calculated for the previous layer. A thread sets the first four bits of the cube code and then shifts the code four bits to the right at the end. The parallelization enforces to enumerate cube edges not just locally, as depicted in Figure 4.3, but also globally. The proposed algorithm prevent a multiple calculation of intersection points between the iso-surface and cube edges by storing the results in an array with the following order: first all x-edges are stored, then all z-edges, and finally the y-edges between the $0-th$ slice and the $1-st$ slice. Then the algorithm continue with the next slice and the array ends with all z-edges of the $(N-th)$ slice. Algorithm 8 gives an overview of the cube code kernel. Note that Algorithm 7 and 8 omit special cases that must be handled explicitly, for example when there is just one partition and no rest.

When the first kernel calculated the intersections (including gradients) and cube codes, *kernel_generate* produces the mesh with help of the classical lookup-table. The lookup table provides the topology of the surface. With its help patches can be defined and only intersections of cube edges and iso-surface has to be calculated. The result is a level set $I_\rho := \{v \in R^3 | f(v) = \rho\}$, which formula can be simplified by subtracting ρ from the whole dataset. At the end this kernel also removes degenerate triangles and feeds the mesh to the second module, the stream simplification. Table 4.1 lists all data structures required during iso-surface extraction.

4.2.3 Parallel Stream Simplification

The simplification module is based on the parallel edge collapse simplification algorithm proposed in the section 3.2 which was originally developed to process a single input mesh in-core. Figure 4.4 gives an overview of the extensions and modifications necessary to

buffers	bytes per entry
	marching cubes
slices	$(dimX \cdot dimZ)N$
gradients	$12(dimX \cdot dimZ)N$
voxel edges	$24((dimX \cdot dimZ)(3N-1)-(dimX+dimZ)N)$
cube code	$(dimX-1)(dimZ-1)(N-1)$
triangles	$60(dimX-1)(dimZ-1)(N-1)$

Table 4.1: Memory consumption and data structures required for the iso-surface extraction. $dimX$, $dimY$ and $dimZ$ are the size of the input grid and N is the number of slices.

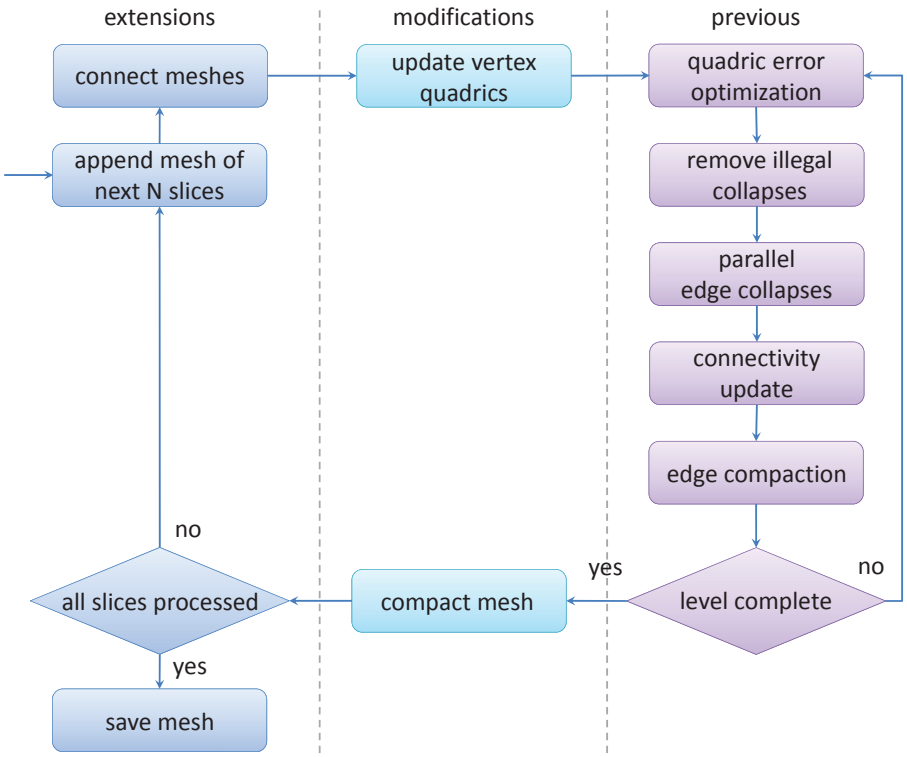

Figure 4.4: Simplification including the extensions (left) and modifications (middle) of the previous simplification algorithm (right).

implement a streaming out-of-core simplification running on the GPU.

First, the mesh build from N slices is transferred from extraction module. Then the vertex quadrics are computed for all new and all connecting vertices. The edge data structure is filled as in the original simplification algorithm. Afterwards, the parallel simplification loop starts. First, the quadric error is optimized and illegal collapses are removed. The edge can collapse, if none of its vertices is on the processing front. In addition, collapses of edges directly connected to these boundary edges are also not possible since the algorithm do not know their local ordering yet. After removing all illegal collapses, the operations can be applied. Finally, the collapsed edges are removed after updating the

face and edge connectivity. If no further collapses are possible, the next partition is added to the simplified mesh. During compaction, the positions of the vertices inside the vertex buffer can change. Therefore, the new positions need to be stored in a lookup table to connect the mesh with the next slices. If no further slices are available, the simplified mesh can be rendered or stored on disk. Note that despite the modifications, the total memory consumption is the same as that of the original simplification algorithm. In addition, the temporary memory required for the extraction module is larger that the total memory required for simplification.

4.2.4 Results

The test system consists of a 3.333 GHz Intel Core i7-980X CPU with 6 GB DDR3-1333 main memory and an NVIDIA GTX580 (841/4204MHz). CUDA is used to implement the parallel simplification and reconstruction. To produce the rendering OpenGL is used. As input different models of 'The Volume Library' [Roe] are used. Table 4.2 gives an overview of the volume datasets which is used for evaluation. The largest volume is the Porsche model car. It also produces the most faces and thus consumes the highest amount of resources. The volumes bonsai tree and CTA-head have a medium size. The CTA-head is chosen to evaluate the proposed algorithm on medical data sets. The bonsai contains many ramifications, which yields in a high number of faces in the simplified model.

model	grid-dim (x,y,z)	file size (MB)
bonsai #2	512 × 189 × 512	48.384
CTA head	512 × 120 × 512	30.720
Porsche	559 × 347 × 1023	193.784

Table 4.2: The dimension and file size of the models which are used. The pvm-format of [Roe] is converted to a raw-file.

Table 4.3 shows the relative and absolute number of cubes crossed by the iso-surface for different iso-values. The number of generated triangles is roughly twice the number of crossed cubes.

model	% Xed cubes	# Xed cubes	# faces
bonsai #2 (20)	4.488921	2,203,645	4,405,952
bonsai #2 (25)	2.293320	1,125,808	2,252,046
bonsai #2 (50)	0.671935	329,858	658,158
CTA head (50)	3.078006	956,441	1,913,256
CTA head (60)	9.529183	2,961,041	5,878,764
CTA head (250)	2.235819	694,745	1,392,432
Porsche (14)	2.404524	4,744,499	9,580,084

Table 4.3: Relative and absolute number of crossed cubes depending on the iso-value, given in parenthesis. In addition, the number of generated faces before simplification is shown.

As partitioning, a fixed size of 12 slices per iteration of the algorithm is used. By comparing the data of the Table 4.4, it can be seen that the extraction time is almost linear in the number of cubes and slightly increases with a higher number of generated faces. The processing performance ranges from 12.0M (CTA head, 60) to 12.9M (Porsche, 14) cubes per second and the number of generated triangles lies between 174k (bonsai #2, 25) and 2.24M (CTA head, 60) per second. The memory consumption is also dominated by the partition size with a small overhead for the simplified mesh. Notice that there is no significant difference in processing times or memory consumption between medical and other data sets. The generated meshes are shown in Figure 4.6 and 4.7.

model	extr.	simp.	# faces	mem.
bonsai #2 (20)	4.09s	5.74s	4,396,060	480.5 MB
bonsai #2 (25)	3.99s	2.86s	2,245,412	430.6 MB
bonsai #2 (50)	3.78s	0.95s	658,158	394.7 MB
CTA head (50)	2.58s	1.19s	1,845,976	423.4 MB
CTA head (60)	2.62s	5.31s	1,216,734	514.2 MB
CTA head (250)	2.58s	1.37s	1,392,432	411.5 MB
Porsche (14)	15.34s	21.12s	4,140,690	923.2 MB

Table 4.4: Computation time for surface extraction and simplification, number of faces after simplification and maximum memory consumption.

The CTA head dataset with an iso-value of 250 has a similar percentage of crossed cubes as the old bone (2.04%) in [ACSE05] and contains a similar medical structure. The CPU used by Attali et al. (1.4 GHz) was approximately three times slower than of used test system (3.333 GHz). Processing the old bone would thus require approximately 7.64 seconds on the used test system, which is roughly three times slower than the proposed extraction algorithm (2.58 seconds). Considering that the total number of cubes is 87.5 percent higher, the speedup is approximately by a factor of 5.6. Comparing the total time, the improvement of the proposed method is even higher. Following the same argument as for the extraction, the total speedup is almost tenfold.

Figure 4.5 analyzes the total memory consumption when processing the Porsche model in detail. For each partition, the maximum memory consumption during extraction and simplification along with the number of faces after simplification is plotted. While the memory gradually increases with the size of the generated mesh, it is dominated by the data required to process the current partition.

When considering mesh quality, the proposed method does not require a time lag being a trade off between memory consumption and quality. Instead, the proposed method guarantees that the ordering of collapse operations is locally preserved and thus produce meshes of the same quality as the underlying simplification algorithm. In contrast to Attali et al. [ACSE05], the directional bias is for example close to zero.

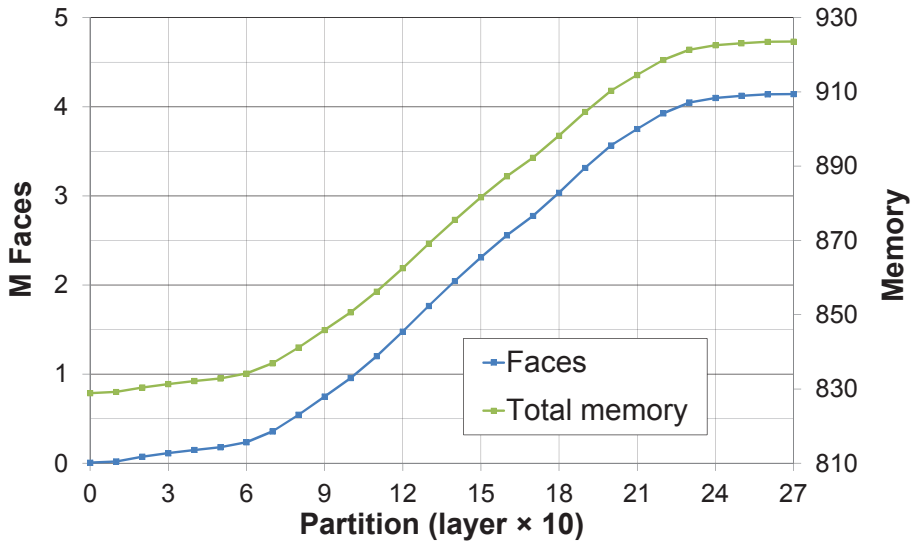

Figure 4.5: Total memory consumption and number of faces contained in the extracted after each partition of the Porsche model was processed.

4.2.5 Conclusion and Limitations

The proposed algorithm shows superior performance and mesh quality compared to previous approaches. Especially removing the need for an explicit time lag by locally blocking simplification operations allows to improve the mesh quality while even reducing the memory overhead. Due to the massively parallel implementation, the proposed approach directly benefits from the future improvements of graphics hardware or other parallel systems.

By guaranteeing the same local order of collapse operations as a simplification of the complete mesh, the same quality as the underlying simplification algorithm is achieved. This also implies a better mesh quality since no artifacts at partition boundaries are introduced.

Currently the proposed implementation is limited to models for which at least three slices fit into memory. The reason for this is that a simple plane sweep partitioning is used. For very large volume datasets it would be possible to use a regular grid instead without changing the core algorithm. The only part that needs to be modified in this case is the connection of the new partition's mesh with the currently simplified one.

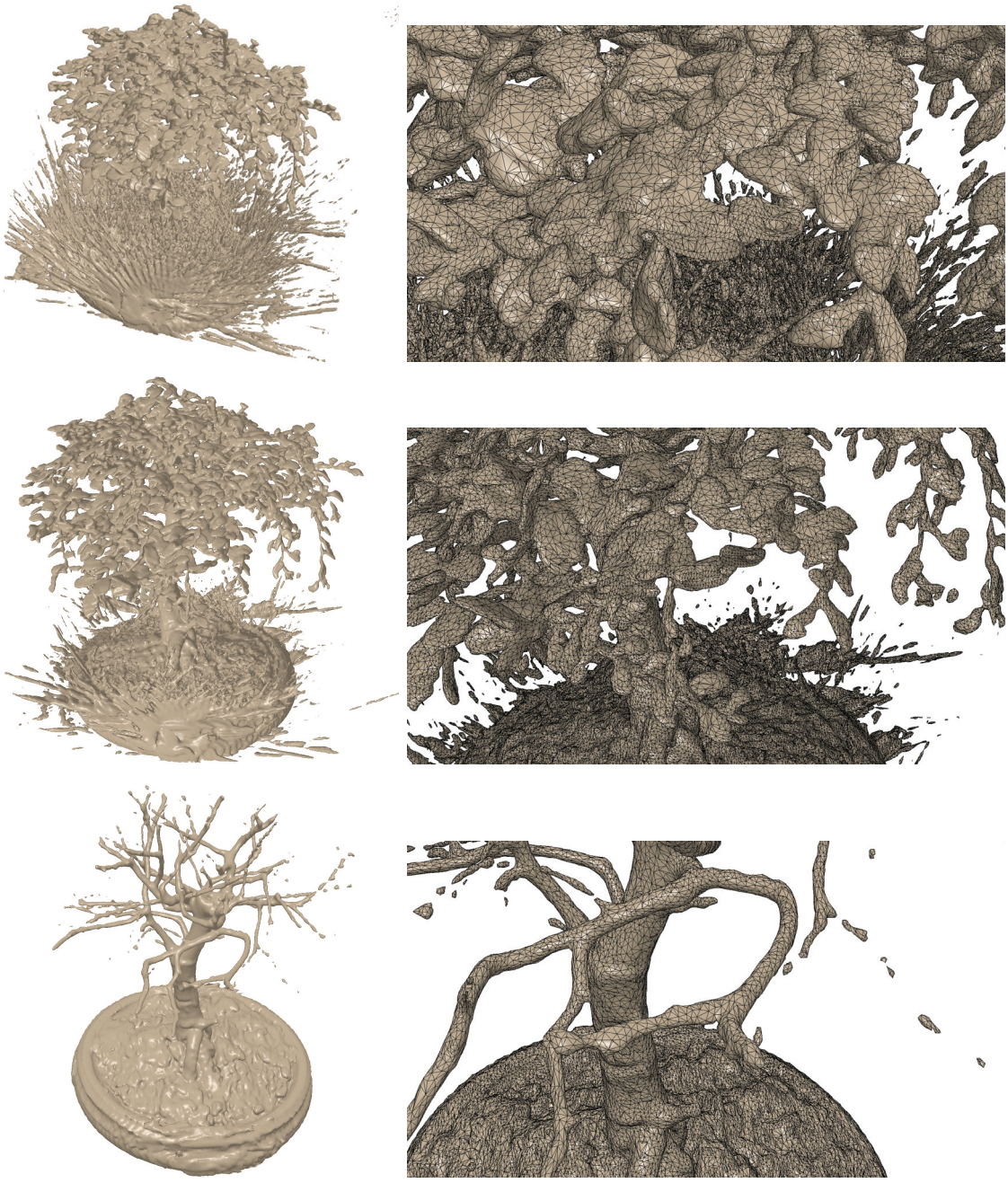

Figure 4.6: Renderings of the extracted and simplified meshes of bonsai #2 (iso 20, 25 and 50). The images on the right show closeups with the mesh overlaid as wire frame.

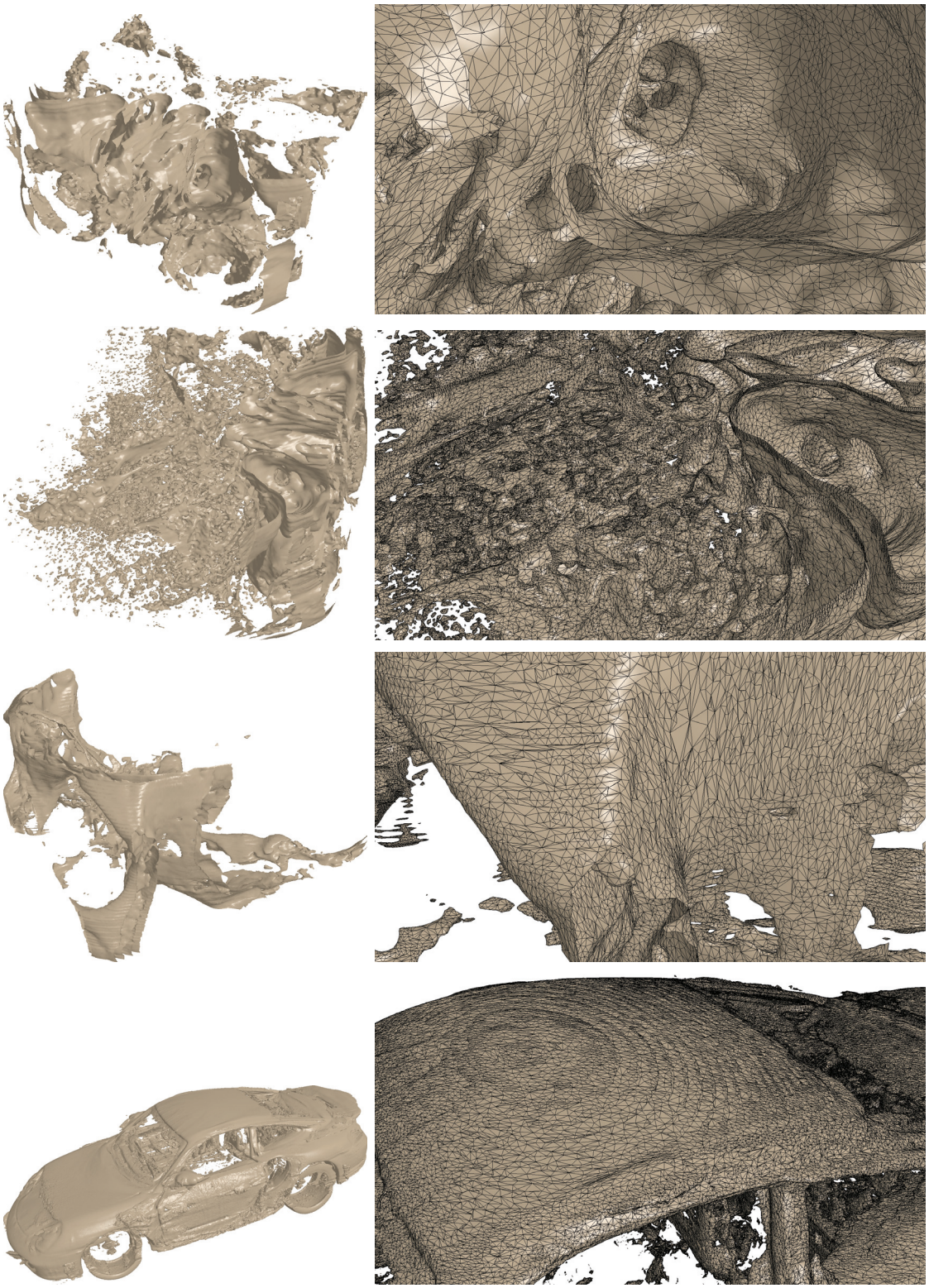

Figure 4.7: Renderings of the extracted and simplified meshes of CTA head (iso 50, 60 and 250) and Porsche (iso 14). The images on the right show closeups with the mesh overlaid as wire frame.

CHAPTER 5

Progressive Mesh Rendering

The desire for high quality polygonal models in interactive applications is constantly increasing. Despite the enormous processing power of graphics processors (GPUs), highly detailed models cannot be rendered in real-time. Often they even do not fit into graphics memory since only models with up to 44.7 million triangles using[34] can be stored within a gigabyte. This even drops to 38.3 million if per vertex texture coordinates are used and so on. The standard solution to reduce rendering time are static or dynamic levels-of-detail (LODs). While static LODs are simply a set of polygon meshes, dynamic LODs store a coarse base mesh and a sequence of refinement operations. Dynamic LODs have the advantages that view-dependent adaption is possible and transitions between LODs, so-called popping artifacts, are much less visible. The most common data structure used in this context are progressive meshes. Sequential algorithms can however not process enough data to fully feed the GPU and the problem of all previous parallel approach are the local vertex dependencies. While this is unproblematic for serial algorithms, the dependencies drastically reduce the number of parallel operations. Thus they do not only increase the number of triangles but also reduce the adaption speed. A high adaption speed is on the other hand the only way to prevent popping artifacts since even prefetching algorithms cannot compensate slow adaption for more than a few frames.

Out-of-core techniques were developed as LOD techniques normally increase the total memory consumption. For static LODs, the model is typically partitioned using a spatial hierarchy. Then a single LOD is generated for each node. This results in a hierarchical LOD (HLOD) structure where only the currently required nodes need to be kept in memory. The approach can also be extended using dynamic LODs for each node for fully view-dependent adaption. In any case, special care must be taken at the boundaries between nodes to prevent visible holes in the model. Out-of-core techniques only shift the problem of limited fast memory to limited bandwidth of slower external devices. Compression techniques are widely used to reduce the bandwidth. Unfortunately, efficient compression approaches

34 Using 32 bit floats.

provide only coarse grained random access. For HLODs, the problem can be circumvented using node-wise compression. The contents of each node are compressed separately and decompressed during loading. The compression however cannot be used to reduce the graphics memory consumption.

The algorithms proposed in this chapter solve the problems discussed above by introducing a novel progressive mesh data structures that are specifically designed for real-time parallel adaption on the GPU.

In a section 5.2, a compact in-core data structure for progressive meshes is presented. It is optimized for parallel processing and low memory consumption on the GPU. The main contributions are:

- A novel random access data structure for progressive meshes that requires less than 50% of an ordinary mesh.
- A massively parallel adaption algorithm running on the GPU that is almost as fast as rendering the adapted mesh. The approach outperforms previous techniques by almost an order of magnitude.

In a section 5.3, a new out-of-core algorithm for real-time view-dependent rendering of huge models is presented, which combine the advantages of view-dependent progressive meshes and HLODs. Using a spatial hierarchy the previous in-core algorithm (Section 5.2) is extended to support out-of-core rendering. In addition a compact data structure for progressive meshes is presented, optimized for parallel GPU-processing and out-of-core memory management. The main contributions are:

- A view-dependent out-of-core progressive mesh data structure which can also be used for occlusion culling.
- No simplification constraints between nodes of the spatial hierarchy.
- A massively parallel adaption algorithm with stable, real-time frame rates.

The in-core (Section 5.2) and out-of-core (Section 5.3) algorithms presented above are suitable for real time rendering of large models, but they have severe limitation. Some splits are postponed several frames as they are waiting for others to be applied before them. This neighborhood dependencies are very problematic for fast panning over the model, because of visible popping artifacts. The same problem have all previous approaches. To solve this problem a novel algorithm for real-time view-dependent rendering of gigabyte-sized models is proposed in a section 5.4, which use a less restrictive dependency scheme and combine the advantages of both previous algorithms. It is based on a neighborhood dependency free progressive mesh data structure. Using a per operation compression method, it is suitable for parallel random-access decompression, without storing decompressed data. By using an optional bounding volume hierarchy, it is suitable for in-core and out-of-core rendering. The main contributions are:

- A view-dependent in-core and out-of-core full random access compressed progressive mesh data structure.
- No inter-dependencies between adjacent vertices to prevent waiting for dependent operations.
- A massively parallel adaption algorithm with stable, real-time frame rates.
- A bounding volume hierarchy for out-of-core rendering and occlusion culling.

5.1 Previous Work

View-dependent simplification has been an active field of research over the last two decades.

5.1.1 Progressive Meshes

In addition to the existing static LOD techniques [GH97b], Hoppe [Hop96] introduced progressive meshes (PMs) that smoothly interpolate between different levels-of-detail. Depending on the view position and distance, a sequence of split- or collapse operations can be performed for each vertex to generate a view-dependent simplification. The inter-dependency of split operations can either be encoded explicitly [XV96] or implicitly [Hop97]. Depending on the view position and distance, a sequence of split- or collapse operations can be performed for each vertex to generate a view-dependent simplification [XV96, Hop97]. Hoppe later optimized the data structures and improved the performance of the refinement algorithm [Hop98]. El-Sana et al. [ESV99] prevented fold-overs of triangles by introducing a view-dependent tree containing modified rules for the operations. The dependency also requires no additional memory, but a suitable neighborhood data structure for the adapted mesh. In addition, the split operations cannot be stored as compactly as with the approach of Hoppe. De Floriani et al. [DFMP98] used multi-triangulations (MTs) based on vertex insertion and removal and proposed a direct acyclic graph (DAG) to the reduce memory. Unfortunately, updates in the MT are computational more expensive than vertex split and collapse operations. Pajarola and Rossignac [PR00] introduced compressed progressive meshes, where the input mesh is simplified in batches. A batch is created by selecting the first 11% of non-adjacent edges from a priority queue. All of these edge-collapses have to be performed in parallel. This allows for a very compact coding, but view-dependent adaption is impossible. Pajarola et al. [PR00] introduced compressed progressive meshes, that allow for a very compact coding, but view-dependent adaption was impossible. Pajarola and DeCoro [Paj01, PD04] developed an optimized sequential view-dependent refinement algorithm. Their *FastMesh* is based on the half-edge data structure and manages split-dependencies by storing a collapse-operation for each half-edge. Diaz-Gutierrez et al. [DGGP05] proposed a hierarchyless simplification

algorithm that can also be used for stripification and compression. While they completely remove any inter-dependency of split operations, an efficient view-dependent adaption is not possible since that requires a split-hierarchy. This however requires 24 additional bytes per vertex of the adapted mesh. In the case of terrains, where vertices lie on regular 2D grids, Levenberg [Lev02] proposed a method that is predicated on fine-grain mesh updates and coarse-grain updates. The GPU-based approach by Ji et al. [JWLL06] generates an LOD texture atlas by resampling the original model onto a regular remesh over a polycube map. They use a vertex shader to displace invisible vertices to infinity. This however has a significant impact on performance since no vertex transformations are saved. DeCoro and Tatarchuk [DT07] propose an octree-based vertex clustering for real-time simplification on the GPU. Adaptive simplification is supported by warping the input mesh. While the algorithm is fast enough to generate LODs at runtime, the visual quality is suboptimal due to the primtive vertex-clustering. Hu et al. [HSH09] proposed a parallel adaption algorithm for progressive meshes. They introduced a relatively compact explicit dependency structure that allows to group vertex splits and half-edge collapses into parallel steps. The drawbacks of this technique are the explicit dependencies that need additional memory and that only half-edge collapses are supported.

5.1.2 Hierarchical Level of Detail

The first HLOD approach was proposed by Erikson et al. [EMB01]. The problem of this technique is that no simplification along cuts between hierarchy nodes is possible without introducing visible gaps. Constraining the simplification however leads to a significant increase of the number of primitives and thus low frame rates. Guthe et al. [GBK03] solved this problem by first using an unconstrained simplification of the nodes. The gaps are then filled during rendering using line strips. Cignoni et al. [CGG*04] proposed a different solution by creating alternating diamond shaped hierarchies. This way the triangles along a node boundary can be simplified at coarser levels. Finally, Borgeat et al. [BGB*05] proposed to use geomorphing to simplify the triangles along node boundaries during rendering. Unfortunately, the transform performance is approximately halved this way such that the previous two approaches are faster. On the other hand, popping artifacts are drastically reduced due to smoother LOD transitions. Another approach are the FarVoxels [GM05], which replace pixel sized triangles by a point and use an octree for point clustering. Sander et al. [SM06] proposed an algorithm that performs geomorphing on the GPU to render a given mesh. This approach extends the idea of Borgeat et al. and applies geomorping on all triangles. The clustered hierarchy of progressive meshes (CHPM) approach [YSGM04] was the first to combine HLOD and progressive meshes. A progressive mesh is stored for each node to allow for smoother LOD transitions. Nevertheless, fully view-dependent adaption is still not possible due to the use of view-independent adaption inside each node.

5.1.3 Compression Approaches

Mesh compression approaches have good compression rates [TR99, AAR05], but random access is not possible. The first approach allowing random access was introduced by Choe et al. [CKL*04]. Kim et al. [KCL06] provide a more effective approach for random access compression, based on their multi-resolution data structure [KL01]. Yoon et al. [YL07] use streaming mesh compression to improve the compression rate over previous approaches. The data are divided into blocks and each block is compressed separately. The approach of Choe et al. [CKLL09] is similar to [CKL*04] but contains some improvements. The performance of this approach was slightly improved by Du et al. [DJCM09] using a k-d tree. The approach of Courbet et al. [CH09] has a slightly better performance but uses a single-rate compression scheme. The CHuMI Viewer [JGA09] introduces a primary hierarchical structure (nSP-tree) in which a kd-tree is embedded to improve the performance. Although all previous compression approaches have good compression rate, only coarse grained random access is supported and interactive rendering is not possible without severe popping artifacts.

5.2 Parallel View-Dependent Refinement of Compact Progressive Meshes[35]

The proposed view-dependent refinement algorithm is based on the vertex hierarchy of progressive meshes [Hop97]. The construction of the split hierarchy is unmodified, but instead of directly using the quadric error (Sections 3.1.2 and 3.2.1) for the LOD selection, the appearance error of Guthe et al. [GBBK04] is utilized to support arbitrary vertex attributes, because it significantly improves the visual quality at the cost of a slightly higher primitive count.

5.2.1 Overview

The progressive mesh is generated by simplifying the original mesh to the base mesh with a series of collapse operations. The original mesh can then be reconstructed by applying the corresponding split operations in reverse order. A view-dependent reconstruction can be generated by performing only those splits that are necessary for the current view point. During this process the *local* ordering of operations needs to be preserved. This leads to the dependency rules formulated by Hoppe [Hop97]:

- The ordering of operations applied to a single vertex must be preserved.

35 In proceedings of Eurographics Symposium on Parallel Graphics and Visualization (EGPGV2010) [DMG10a], based on the diploma thesis [Der09].

- A split can only be applied if the next split operation of each neighboring vertex was generated earlier during simplification.
- Edge collapse operations are only legal if the next collapse of each neighboring vertex was created later.

The first dependency rule can be efficiently encoded in a forest of binary trees where the root nodes are the vertices of the base mesh. Figure 5.1 shows an edge collapse operation col_v which removes the vertex v_u and modifies v. The adjacent faces f_l and f_r of v and v_u degenerate and are removed from the mesh. The corresponding vertex split spl_v inverts this operation. Accordingly the faces f_l and f_r are generated when the vertex v is split into v and v_u. In addition, some of the faces adjacent to v become adjacent to the new vertex v_u.

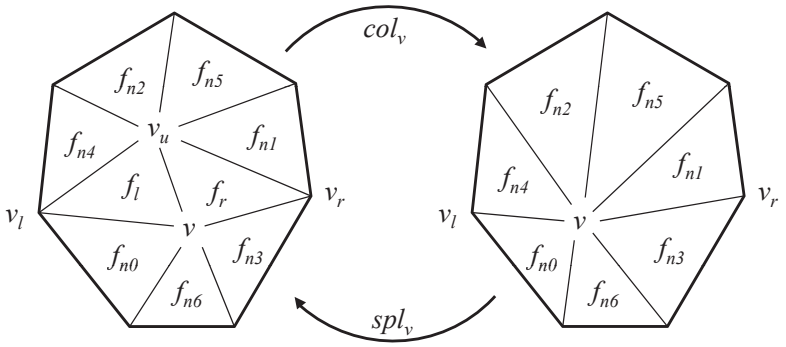

Figure 5.1: Edge collapse and vertex split operation.

5.2.1.1 Tree Structure and Dependency Coding

The operations are stored in a one-dimensional array. The tree structure could now be encoded by storing the indices i of the left and right child for every node. This would however require 8 bytes per operation, whereas a binary tree can be encoded using only two bits per node with a succinct coding. It is only store if left and right child are present using a single bit for each of them in a *tree structure byte*. Then the indices i_l and i_r are calculated from the current index i:

$$i_l = 2i + n_r - skip_i \tag{5.1}$$

$$i_r = 2i + n_r - skip_i + 1, \tag{5.2}$$

where $skip_i$ is the number of empty nodes up to the left child and n_r is the number of root nodes (i.e. base mesh vertices). An example of the operation tree is shown in Figure 5.2. Unfortunately, calculating the current $skip_i$ requires parsing the complete data structure up to the current node and counting the number of zero bits. Instead of this, the skip count is stored with every operation. This would again require 4.25 bytes per operation. A

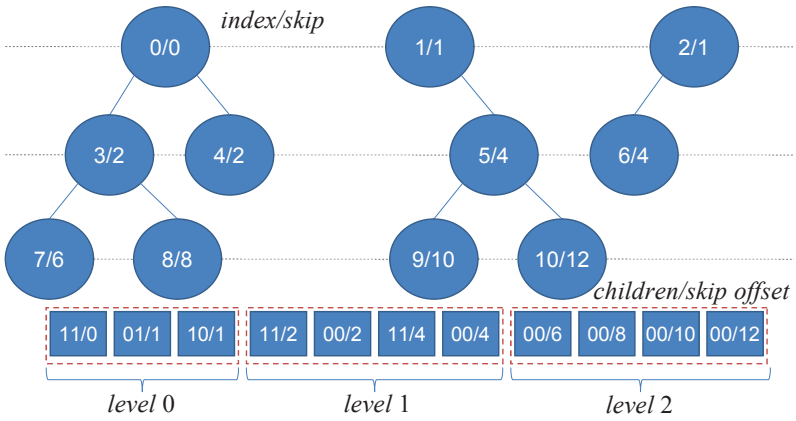

Figure 5.2: Compactly encoded forest of binary trees.

far more compact encoding with equal computational complexity can be achieved by only storing the skip for every n-th node. Then the number of zero bits is counted from the last *base skip*. In the proposed approach, the counting is completely avoided by encoding the difference between $skip_i$ and the base skip. This difference is stored the six remaining bits of the tree structure byte. In these six bits the numbers up to 63 can be encoded. Considering the fact that $skip_i$ can be at most two larger than $skip_{i-1}$, the base skip for every 32nd node need to be stored. This sums up to a total of 1.125 bytes per node to encode the tree structure.

Unfortunately the operation indices cannot be used to preserve the local ordering since the tree encoding has changed their sequence. This necessitates to explicitly encode the neighborhood dependency for each operation. A compact encoding of this dependency can however be derived rather simple: the algorithm start with the base mesh M_0 and collect all operations that can be applied directly. After applying these, the algorithm get the maximally refined next mesh M_1. Repeating this procedure, a series of meshes $\{M_0, M_1, \ldots, M_n\}$ is generated, where M_n is the original mesh. A split that refines M_i to M_{i+1} is then defined as having a split level of i. The local ordering is preserved if only splits with a lower split level than those of the neighboring vertices are applied. The corresponding condition for an edge collapse is simply that the split level is less or equal for all vertices adjacent to v and v_u. As the level n of the original mesh is proportional to the logarithm of its vertices, a single byte is sufficient to store the split level even for models with several millions of triangles.

5.2.1.2 Topology Encoding

To efficiently encode which of the neighbor vertices are adjacent to the new faces, an ordering on the vertex neighborhood is imposed. Then the algorithm can simply encode the vertices v_l and v_r by their rank in this ordered sequence. The same applies to the partitioning of the neighbor faces into those that are adjacent to v_u after the split and

those that remain adjacent to v. Here a bit vector can simply used where an entry is set to one of the v is replaced by v_u for that face. Previous approaches use a cyclic ordering that requires a neighborhood graph. Instead, a unique ID is assigned to every vertex and face which remains constant over any modification of the mesh. This allows for an efficient handling of vertex and face orderings and enables the algorithm to support non-manifold meshes. While the unique ID's of base mesh vertices and faces are simply their indices, the ID's of vertices and faces created by a split operation are defined as follows:

$$ID_{v_u} = v_0 + i_s \tag{5.3}$$

$$ID_{f_l} = f_0 + 2i_s \tag{5.4}$$

$$ID_{f_r} = f_0 + 2i_s + 1, \tag{5.5}$$

where v_0 and f_0 are the number of vertices and faces in the base mesh and i_s is the index of the split operation.

Since a split creates zero to two faces, the algorithm also need to encode the case that f_l or f_r are not present. If both ranks are encoded using four bits each, up to 15 neighboring vertices can be handled within a single byte. When storing this partitioning in two bytes, vertices with up to 16 neighboring faces are supported. The limitations to 15 neighboring vertices and 16 faces do not impose significant restrictions since the average valence in a triangle mesh is 6 and valences above 15 are extremely rare. Nevertheless, the simplification algorithm needs to adhere to these restrictions. If the original mesh contains vertices with higher valence, they can not be collapsed until enough adjacent vertices are removed.

5.2.1.3 Attribute Encoding

In addition to the connectivity, the new attributes (position, normal, texture coordinates, etc.) of v and v_u must also be stored. Both of them are encoded as difference to the attributes of v before the split. In contrast to previous approaches a linear quantization of the whole input data is not used but an individual one for each operation. This allows to reduce the quantization error by inserting a dummy split that does not change the connectivity. As the same quantization is used for all attribute differences of the operation, they must be scaled relatively to each other. Since the overall quantization error needs to be minimized, the relative scale factors should be as close as possible to the absolute difference values of all compression operations. This is achieved by computing the mean absolute difference of each attribute difference over all split operations. Then these norms are used as scale factors. As the scaling is relative to the other attributes, all of them can multiply with an arbitrary factor. This factor is chosen such that the maximum absolute difference is scaled to MAX_HALF in order to utilize the complete range of values for the scaling of each operation. This per operation scaling is a division by a factor s such

that the difference values are mapped into the interval $[-1 \dots 1]$. The optimal scaling (the minimal s) is then stored as half precision floating point. Despite the adaptive scaling one difference is most of the time significantly larger than the others. To accurately represent small and large values a cubic function is applied before the final quantization to n bits. Given a discrete value $d \in [-1 \dots 1]$, the quantized value q is:

$$q = \sqrt[3]{d}(2^{n-1} - 1), \tag{5.6}$$

where n is the number of bits. The dequantization is:

$$d = \frac{q^3}{2^{n-1} - 1} \tag{5.7}$$

5.2.1.4 Refinement Criteria

Three view-dependent criteria determine whether a vertex needs to be split or can be collapsed. A vertex can be collapsed if it is either outside of the view frustum or its normal if facing away from the viewer. Since each vertex of the adapted mesh represents several original vertices, there exists no single normal. Instead, the maximum angular deviation α from the normal of the simplified vertex is encoded. Given the view direction $\mathbf{d} = \frac{\mathbf{p}-\mathbf{e}}{\|\mathbf{p}-\mathbf{e}\|}$, where $\mathbf{p}$ and $\mathbf{e}$ are the vertex and eye position, and the normal $\mathbf{n}$, the vertex is back-facing if:

$$\mathbf{n} \cdot \mathbf{d} > \sin\alpha \tag{5.8}$$

To save memory $\sin\alpha$ is not stored as floating point value but only a quantized value using four bits. During quantization the ceiling is used such that the inaccuracy only leads to back-facing vertices being classified as front-facing. Note that even the quantization to four bits, as it is used in the implementation, only marginally increases the number of rendered triangles. The vertex can possibly be collapsed when it is back-facing or outside the view frustum. Otherwise the algorithm need to check the simplification error of the associated split and collapse operations. The projected simplification error ϵ_s of the vertices' split operation s consists of the geometric distance γ_s and the attribute difference μ_s. The maximum simplification error δ_s is then simply $\max(\gamma_s, \mu_s)$. While the attribute difference is equal for all view directions, the projected geometric error depends on the angle between $\mathbf{n}$ and $\mathbf{d}$. The squared projected error can now be written as:

$$\epsilon_s^2 = \frac{((\mathbf{n} \cdot \mathbf{d})\mu_s)^2 + (||\mathbf{n} \times \mathbf{d}||\delta_s)^2}{D^2}, \tag{5.9}$$

where $D = \|\mathbf{p} - \mathbf{e}\|$ is the view distance. To efficiently store the two simplification errors, the fact is exploited that δ_s is more important as it dominates the projected error. So instead of two floating point values, only δ_s is stored as half float and quantize the ratio

$\lambda_s = \frac{\mu_s}{\delta_s}$ using four bits. The ratio and $\sin\alpha$ are stored into a common byte. Then the projected error is written as follows:

$$\epsilon_s^2 = \frac{((\mathbf{n}\cdot\mathbf{d})\lambda_s)^2 + ||\mathbf{n}\times\mathbf{d}||^2}{D^2}\delta_s^2 \tag{5.10}$$

$$= \frac{(\mathbf{n}\cdot\mathbf{d})^2(\lambda_s^2 - 1) + 1}{D^2}\delta_s^2 \tag{5.11}$$

If ϵ_s exceeds a given threshold the split operation has to be applied. On the other hand, the vertex can be collapsed, if the simplification error ϵ_c of its collapse operation c is below the threshold. In the current implementation, $\tan\frac{1}{60}^\circ$ is used as threshold to guarantee that the difference is not perceived if the projection matches the real-world view condition. Other thresholds, e.g. $\frac{1}{2}$ pixel screen space error, are also possible.

5.2.1.5 Dynamic Data Structures

The adaption algorithm maintains a static *split tree* storing the split hierarchy as well as a dynamically updated *VertexBuffer*, *IndexBuffer*, and some other temporary data. The SplitTree contains both the topologic and the geometric information of the progressive mesh. The base mesh is only used to initialize the dynamic data and thus not kept in memory. By using these data structures, the selectively refined mesh can be rendered in real-time. Table 5.1 shows the static and dynamic data structures in detail, which are required to maintain the relevant buffers. Since the complete algorithm runs on the GPU, the all data is stored in graphics memory.

The main data structures required for rendering are the *vertex buffer*, which contains the position and attributes of the adapted vertex and the *index buffer* that contains the connectivity. Both are stored as vertex buffer objects (VBOs) and are therefore separated from all other data. The neighborhood information for the currently applied split operations is stored in the neighborhood array. It contains the number of adjacent triangles and their indices for each vertex that will be split. Since up to 16 neighbor triangles exist, 68 bytes per split are needed. As only a quarter of the vertices can be split in parallel, this translates to 17 bytes per active vertex. For each active vertex v, the algorithm additionally stores its state, the unique ID and the next split and collapse operation.

As the tree structure of the progressive mesh is only stored from root towards the leaves, the upwards references are kept in the dynamically updated *collapse tree*. Its elements consist of the index of the corresponding split operation i_s, a reference to the previous collapse, and a reference to the vertex v_u that is removed by this operation. In addition, each active vertex v holds a reference to the corresponding entry in the collapse tree. The complete structure is shown in Figure 5.3. In addition, three temporary buffers are required for the compactions: one for the scan input, one for the output and one temporary

buffers	elements	memory (bytes)
static structures		
operations	tree structure	$1\frac{1}{8}n$
	dependency	$1n$
	ref. criteria	$3n$
	topology	$3n$
delta vectors	quant. delta	$2kn$
	delta scale	$2n$
dynamic structures		
active faces	index VBO	$24m$
	triangle ID	$8m$
active vertices	vertex VBO	$4km$
	vertex ID	$4m$
	next split	$4m$
	next collapse	$4m$
	state v_{state}	$1m$
collapse tree	split index i_c	$4m$
	prev collapse	$4m$
	vertex v_u	$4m$
temporary	prefix sum	$24m$
neighborhood	size	$1m$
	triangle index	$16m$
total		$(10\frac{1}{8} + 2k)n + (98 + 4k)m$

Table 5.1: Elements of the data structure. k, n, and m are the number of attributes, original, and base mesh vertices.

buffer [SHZO07]. Each buffer contains four bytes per entry and the maximum number of entries is the number of triangles in the current mesh which is twice the number of vertices.

Most other algorithms support meshes with $k = 8$ attributes only consisting of position, normal, and 2d texture coordinates. With $k = 8$ the complete hierarchy and adapted mesh use a total of $26\frac{1}{8}n + 130m$ bytes, where n and m are the numbers of vertices in the original and adapted meshes respectively. Table 5.2 shows a comparison with previous view-dependent LOD schemes. As for highly detailed models, it is generally impossible to view the whole surface at high resolution within a single frame, one can assume that $m << n$. Therefore, the requirements for the static data structures that are the only depending on n are more important than the ones of the dynamic structures. With only $26\frac{1}{8}n$ the proposed data structure favorably compares to all other view-dependent methods with at least a reduction of 62%, so roughly a third of the memory.

Compared to the 129 bits per vertex (bpv) the proposed algorithm need, the compressed progressive meshes of Hoppe [Hop96] and Pajarola and Rossignac [PR00] require 31–50 and 21–28 bpv only. The drawback of those is that both use variable length coding which is not suitable for efficient random access decompression. Therefore, their approaches

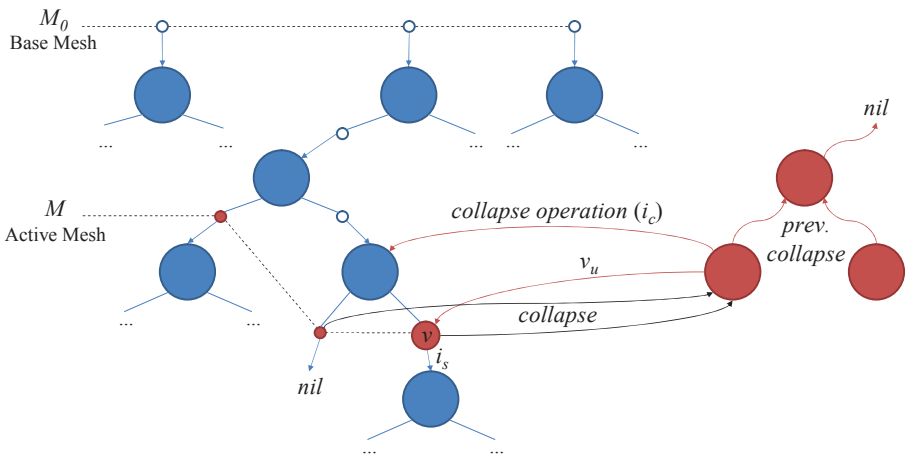

Figure 5.3: Linking between active vertices, the SplitTree, and the CollapseTree.

View-Dependent LOD scheme	*Memory size (bytes)*
VDPM [Hop97]	$216n$
SVDLOD [Hop98]	$88n + 100m$
MT [DFMP98]	$75n$
VDT [ESV99]	$90n$
FastMesh [PD04]	$88n + 6m$
PVDPM [HSH09]	$69n + 56m$
Proposed scheme	$26\frac{1}{8}n + 130m$
Indexed Face Set	$44n$

Table 5.2: Comparison of memory size with previous schemes for $k = 8$ attributes.

cannot be used for selective parallel LOD adaption. In addition, [PR00] does not support view-dependent refinement at all.

5.2.2 Runtime Algorithm

In order to efficiently exploit the parallel architecture of the Compute Unified Device Architecture (CUDA), the algorithm is subdivided into several steps, that are performed for each parallel adaption. The partitioning is chosen such that each step can be processed completely in parallel.

To classify which operation can or should be applied to a vertex, those options are tracked in a status byte v_{state}. Two bit flags, *potential_split* and *potential_collapse*, are used to mark those vertices that can possibly be split and/or collapsed. If their neighborhood prevents any of these operations, the according bit is cleared. The operation that should be applied to the vertex is stored in two other bit flags, *want_split* and *want_collapse*. A combination with the corresponding potential flags marks a vertex for a split or collapse operation. In addition, a vertex can have one of the following two special states after a collapse operation: the vertex that was removed is labeled with the *removed* state and

the other one is marked as *collapsed*. The state is stored in a separate array to facilitate coalesced reading and writing since it is accessed very often to by aligned to four bytes. Based on these states, split and collapse operations are applied. The parallel stages of the algorithm are discussed in the following. The basic phases of the algorithm are shown in Figure 5.4 together with the static and dynamic data they access.

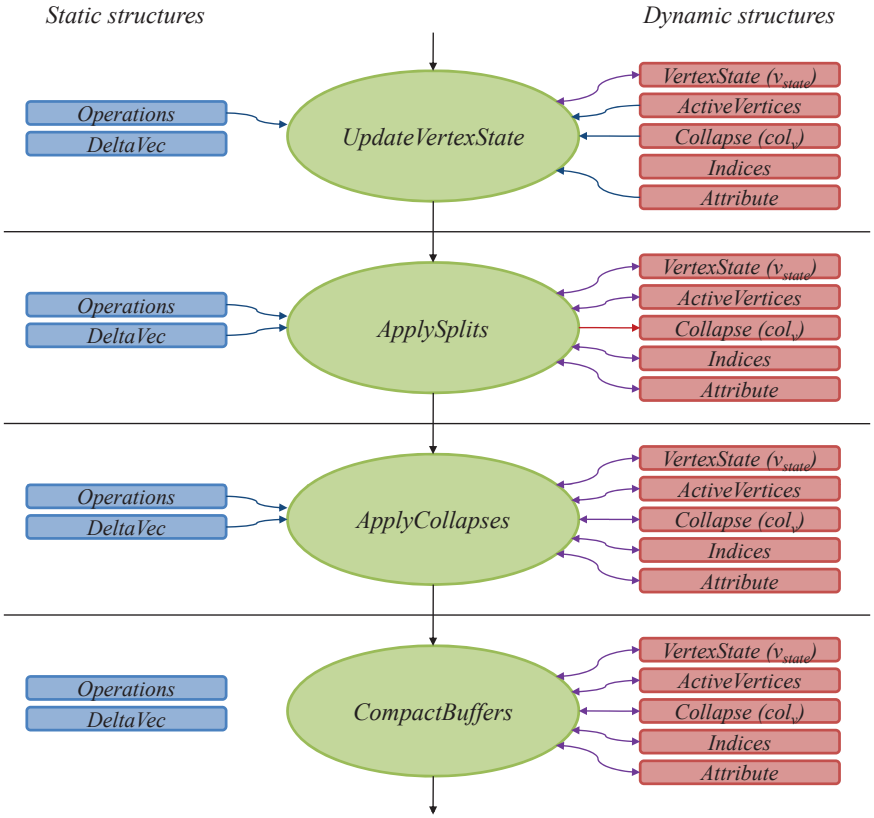

Figure 5.4: Read/Write Access of the individual refinement steps.

5.2.2.1 Vertex State Update

If the refinement criteria determine that vertex v needs to be split, the *want_split* flag is set in its state. Otherwise, the *want_collapse* flag is set if the refinement criteria allow a collapse and the vertex is marked as *potential_collapse*. In all other cases no operation is required for v, unless it was already marked as *want_split* in a previous iteration. In both of these cases the state remains unchanged. After the desired operation is determined for each vertex, the algorithm need to check if both vertices of a collapse operation are marked for coarsening. If only one of them is marked, the *want_collapse* flag is removed again. Note that since the algorithm check for impossible operations later, this step is not strictly necessary but leads to a considerable speedup.

Due to the dependency of split operations the algorithm may have marked vertices for

split or collapse operations that cannot be performed. In case of a collapse this is not problematic since the operation is not necessary to achieve a desired quality. For a split operation however, those neighboring vertices that must be split before the current vertex v need to be found. For this purpose, each face f is checked whether one of its vertices is marked as *want_split* but another vertex of the same face has a lower split level. If this vertex is not already marked for splitting as well, its *want_split* flag also needs to be set. The procedure is performed twice since only the neighboring dependent splits are marked each time. This way, every dependent split with a topological distance of d_t is marked after at most $\frac{d_t}{2}$ adaption iterations. To remove splits and collapses that cannot be performed yet, the algorithm traverses all triangles and again checks the vertex states. For the split operations, only the vertex with the lowest split level can be split in each face f. In addition, if any vertex is marked for splitting, no vertex of f can be collapsed. If no vertex of the face needs to be split, finally the collapse operations are checked. Here only the collapse operation with the highest level can be performed in each triangle. Note that the corresponding *want* flags are not changed, but the *potential* flags to prevent repeated checking of the same vertices. Algorithm 9 shows the complete vertex update partitioned into the four stages described above.

When processing large amounts of data on the GPU the aligned memory access of each thread group (warp) is crucial for performance. This access pattern is called coalesced reading and writing. When looping over all faces, the three vertex indices of each face are needed but directly loading them from global memory would violate coalescing. To prevent this, the algorithm read all vertex indices of a thread block into its shared local memory and then fetch the indices of the current triangle from there.

5.2.2.2 Parallel Vertex Splits

After updating the state of all active vertices and removing illegal splits and collapses, the operations can be applied. Before the splits can be performed, the neighborhood of each split vertex needs to be known. The neighborhood information is collected by traversing all faces and if a face f is adjacent to a split s_v, the face index is added to the neighborhood of v. As the algorithm only want to collect the neighborhood for the vertices that are currently split, a so-called compaction operation on the split indices is performed firs. For this purpose, the parallel compaction algorithm of Sengupta et al. [SHZO07] is used to compute an array of split indices. After generating this array and collecting the face neighbors for each split vertex v, the following operations are performed:

1. The new vertex v_u is generated and v is moved to it's new position.
2. The two faces f_l and f_r are added to the index buffer and the triangle ID array.
3. The other faces in the neighborhood of v are relinked according to the encoded topology changes.

```
foreach vertex v in parallel do
    if marked(v, collapsed)
        mark(v, potential_split, potential_collapse)
    if need_split(v)
        mark(v, want_split)
    elif may_collapse(v) && marked(v, potential_collapse)
        mark(v, want_collapse)
foreach vertex v in parallel do
    v_u = get_other(v)
    if v! = v_u
        if !marked(v_u, want_collapse)
            unmark(v, want_collapse)
        if !marked(v, want_collapse)
            unmark(v_u, want_collapse)
repeat twice
    foreach face f in parallel do
        if any_vertex_marked(f, want_split)
            level_max = get_max_active_split_level(f)
            mark_dependent_splits(f, level_max, want_split)
foreach face f in parallel do
    if any_vertex_marked(f, split)
        level_min = get_min_active_split_level(f)
        unmark_dependent_splits(f, level_min)
    if any_vertex_marked(f, want_split)
        unmark_all_collapses(f)
    elif any_vertex_marked(f, collapse)
        level_max = get_max_active_collapse_level(f)
        unmark_illegal_collapses(f, level_max)
```

Algorithm 9: The four parallel stages to update the vertex states. The third stage is performed twice to speed up the propagation of dependent splits through the mesh.

4. The adjacent vertices of v and v_u are marked as *potential_split* as their split could be waiting for the current one. As the adjacent vertices cannot be collapsed, the *potential_collapse* flag is cleared.

5. v and v_u are marked as *potential_split* and *potential_collapse* since both operations could be possible in the next frame.

Algorithm 10 gives an overview of the complete parallel vertex split procedure.

As the neighborhood information is parsed identically for every split vertex, the coalesced reading is achieved with the following layout: First, the number of adjacent faces is stored, then the indices of the first neighbor triangle, than that of the second and so on. In addition to this layout, the algorithm must assure that each new block of indices begins at an address that is a multiple of 128. Therefore, the number of splits is rounded up to the next multiple of 32 for addressing in the neighborhood array.

```
compact(splits)
foreach face f in parallel do
    if adjacent_to_split(f)
        append_to_neighborhoods(f)
foreach split s_v in parallel do
    split_vertex(v)
    add_faces(v)
    relink_neighbor_faces(v)
    mark_neighbor_vertices(v, potential_split)
```

Algorithm 10: Parallel vertex split algorithm.

5.2.2.3 Parallel Edge Collapses

To perform the collapse c_v, the corresponding vertex v_u is required. Since v_u is stored with the collapse operation, the algorithm simply need to check whether the current vertex is different. The collapse is only applied if both are marked as *collapse*. The operation c_v marks vertex v_u as *removed*, moves v to the target position and marks it as *collapsed*. In addition, the target vertex for v_u is stored in the next split since this is not required any more after removing v_u. Then all faces are relinked by checking if a vertex of face f was removed. In this case, the target vertex is fetched and the vertex of the face is set accordingly. If the face becomes degenerated it is removed as well. When a collapse was applied to one vertex of the face, all other vertices are candidates for a possible collapse and are marked as such. Algorithm 11 shows the parallel processing of the edge collapse operations.

```
foreach vertex v in parallel do
    if marked(v, collapse)
        v_u = get_other(v)
        if v! = v_u && marked(v_u, collapse)
            collapse_vertices(v, v_u)
foreach face f in parallel do
    relink_vertices(f)
    if degenerate(f)
        remove_face(f)
    else if changed(f)
        mark_vertices(f, potential_collapse)
```

Algorithm 11: Parallel edge collapse algorithm.

5.2.2.4 Buffer Compaction

The final step of the adaption is the compaction of buffers where elements have been removed. These buffers are the active vertices (including the vertex VBO), active faces (with the index VBO), and collapse operations. Note that when compacting the vertices or collapses, the references to them must be updated accordingly. While the compaction

of the faces and thus the indices is mandatory since the index VBO is used for rendering, the compaction of the vertices and collapse operations is not. The latter two only need to be compacted every few frames to prevent bloating of the buffers. As only a few elements are removed each time and the ordering does not need to be preserved, a specialized in-place compaction algorithm was developed. In contrast to previous approaches, it has the advantage that it do not need to duplicate the array that is compacted. Otherwise it would need copies of all dynamic data structures except the temporary buffers and the neighborhood information which would drastically increase the memory consumption.

The main idea of the compaction is to first calculate the number n_c of elements after the compaction. Then all gaps before n_c are filled with elements after n_c (see Figure 5.5). First the valid elements are marked with one and the invalid ones with zero. Then the prefix sum is computed using the parallel algorithm of Sengupta et al. [SHZO07]. This gives us the number of valid elements n_c as well as the first valid element that needs to be moved. Then the positions of the empty elements is gathered in the final array. Their position in the *free position* array can be computed by subtracting the prefix sum from their index. Finally, the target position of the elements that need to be moved are computed by subtracting the *first moved* from the prefix sum of the current element. Then the position is looked up in the *free position* array and can copy the element into the compacted buffer. The algorithm does not require additional temporary memory except that used to compute the prefix sum. The free positions can overwrite the flag array, since the flags are not required anymore after computing the sums.

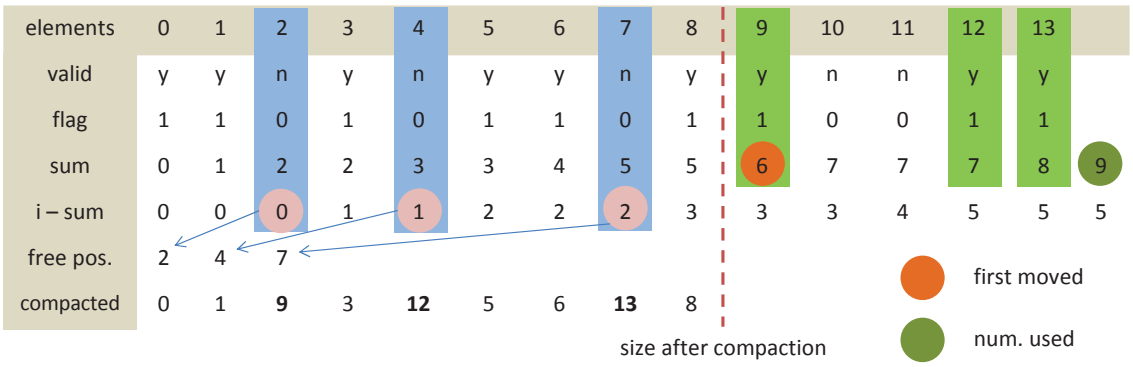

Figure 5.5: Basic principle of the in-place compaction algorithm.

5.2.2.5 Memory Management

During adaption the memory requirements of the dynamic data structures can grow or shrink. To alleviate the cost for memory allocation and copy when the size of an array is modified, more memory is reserved always than currently required. In addition, the array size is restricted to multiples of 4096 elements. Figure 5.6 shows an example of growing and shrinking a data structure. If the currently required amount of memory exceeds the

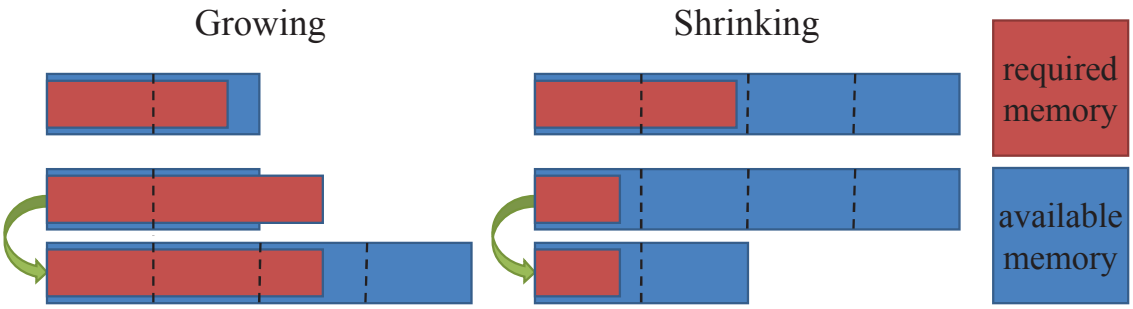

Figure 5.6: Growing (left) and shrinking (right) of an allocated array during adaption.

array size, one additional block is allocated to prevent re-allocation in the next frame. For shrinking a similar strategy is employed by only allocating a smaller array if more than two blocks are empty. Despite reducing the memory consumption, one free block is still keep to prevent quick re-allocation when the array is growing again.

To improve the rendering performance, the triangles are sorted such that the transform cache can be utilized. As the ordering is gradually destroyed when inserting new faces at the end of the buffer, the algorithm need to restore it every few frames. Since the memory consumption also changes when many operations are applied, the sorting is simply integrated into the memory management. When a new buffer is allocated for the index VBO and the face IDs, the triangles are sorted by their minimum index during the copy operation. This results in a mesh that is mainly composed of triangle fans and reduces the transform cost by a factor between of two to three. In total, the rendering time is reduced by 20% to 50% depending on the complexity of the fragment shader. If no allocation occured for more than two seconds, a re-allocation is forced and thus a sorting of the faces. This is necessary as gradual movements do not quickly enough lead to changes in memory consumption but nevertheless perturb the ordering.

5.2.3 Results

The test system is built of a 3 GHz Intel Core2 Duo CPU with 2 GByte of main memory and a GeForce GTX 285. The OpenGL API is used for rendering and CUDA to implement the parallel algorithm. First the memory requirements of the static progressive mesh data structure are evaluated. Table 5.3 gives an overview of the progressive meshes that are used as input and the number of added dummy split operations.

With the exception of the Phlegmatic Dragon, the number of dummy splits is significantly below 1% of the original number of operations. But fortunately, the compression ratio of this model will increase again as it has the highest number of attributes k as tangents and texture coordinates are required to map a BTF on the model. All other models except the manuscript, which additionally stores per vertex color, only use position and normal as vertex attributes. The maximum split level is proportional to the logarithm of the ratio of

model	v_0	f_0	k	*# ops.*	*# dummy ops.*	lvl.
Phl. Dragon	41	44	14	240,016	6,099 (2.54%)	142
St. Dragon	815	536	6	436,830	3,215 (0.74%)	207
Buddha	727	1866	6	542,925	3,529 (0.65%)	135
Manuscript	42	17	10	2,155,575	3,369 (0.16%)	191
Asian Dragon	35	16	6	3,609,565	7,801 (0.22%)	253

Table 5.3: Progressive meshes used as input, number of added dummy split operations, and maximum split level.

original to base mesh vertices. Therefore, larger models can be handled by increasing the complexity of the base mesh. The resulting compressed sizes, compared to an indexed face set that is traditionally stored on the GPU for rendering, are listed in Table 5.4. As expected, the models with higher number of attributes are slightly less compressed by the proposed method. Nevertheless, the memory reduction is relatively similar for all models. The memory consumption lies between 46% and 50% compared to an indexed face set.

model	v_{max}	f_{max}	*mem. IFS*	*mem. PM*
Phl. Dragon	240,057	480,076	18.3MB	9.0MB (49.2%)
St. Dragon	437,645	871,414	19.9MB	9.3MB (46.5%)
Buddha	543,652	1,087,716	24.9MB	11.5MB (46.2%)
Manuscript	2,152,840	4,305,679	123.3MB	57.9MB (47.0%)
Asian Dragon	3,609,455	7,218,906	165.2MB	76.1MB (46.1%)

Table 5.4: Comparison of the static data that resides in graphics memory compared to an indexed face set.

During rendering, the dynamic data structures consume additional memory. For all models except the phlegmatic dragon, the total amount of graphics memory nevertheless stays below that of an indexed face set. Table 5.5 shows the number of rendered faces, the total rendering time, and the memory consumption for the views shown in Figure 5.9. For almost all models the required memory and total frame time are always less than that of the original mesh. The coarsening on the back faces and outside the view frustum can be clearly seen in the external views of the adapted models. The reduced level-of-detail further away from the camera can also be noticed at the example of the Asian Dragon.

Figure 5.7 shows the adaption and rendering time together with the memory consumption for a pre-recorded movement around the Asian Dragon. The consumed graphics memory is always less than required by the original model. The frame rate seldomly drops below the 60 Hz of the display even when a high number of triangles is required. As the time required for each frame is often significantly below 16 ms, more than one adaption iteration could even performed and only render once. Compared to static hierarchical LODs (HLODs) using the same error measure [GBBK04], the number of primitives is reduced by a factor of 3 to 5 and the frame rate improves by a factor between two and three. A special problem

model	*rendered # faces*	*memory (MB)*	*total frame time (ms)*
Phl. Dragon	224,090 (46.7%)	24.1 (129.1%)	10.4 (131.4%)
St. Dragon	190,236 (21.8%)	19.1 (93.3%)	3.2 (95.8%)
Buddha	152,716 (14.0%)	19.5 (78.1%)	3.3 (68.7%)
Manuscript	274,678 (6.4%)	73.5 (59.6%)	4.5 (39.8%)
Asian Dragon	646,844 (9.0%)	108.9 (65.9%)	10.5 (41.2%)

Table 5.5: Memory consumption and total rendering time of the different models. The ratio compared to rendering an indexed face set of the original model is shown in parenthesis.

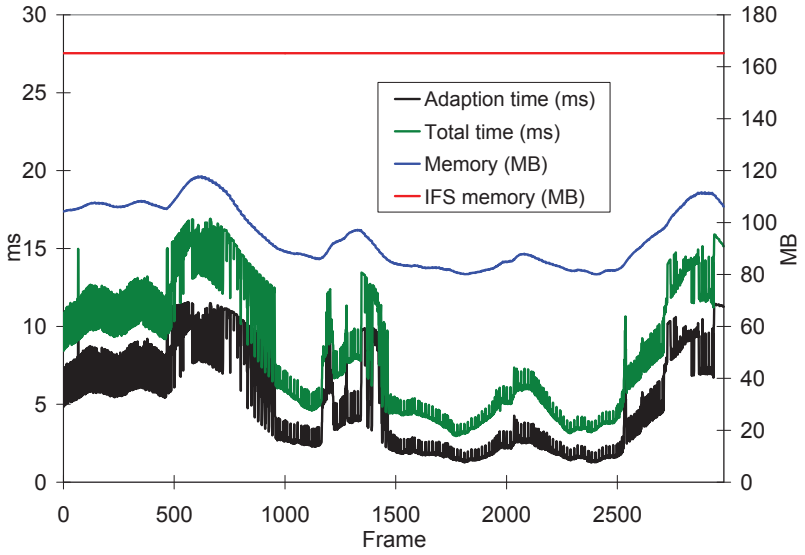

Figure 5.7: Timings and memory consumption for the Asian dragon with a pre-recorded camera path.

of HLODs is that rendering is split into several independent render calls which reduces the number of primitives per second compared to a single mesh.

Figure 5.8 shows a detailed analysis of the runtime of each step of the adaption and rendering. Note that the rendering performance is identical to rendering a static model with the same number of triangles and thus the proposed method needs approximately 2.6 times as long as rendering a static mesh. Considering that the vertices already cut down by half due to the simplification of back faces, the proposed method will almost always be faster than rendering an indexed face set of the original model. While this even holds for rather coarse models, the performance gain increases with the complexity of the original mesh. Due to the time required for the pixel shaders, the speedup is of course not linear with the reduction. On average the algorithm can process 120 million triangles per second (M△/sec). This is a speedup of factor 9.2 compared to the approach of Hoppe et al. [HSH09] that only achieves 13 M△/sec on the used test system. The main source of speedup is probably due to the fact a single CUDA compaction is used

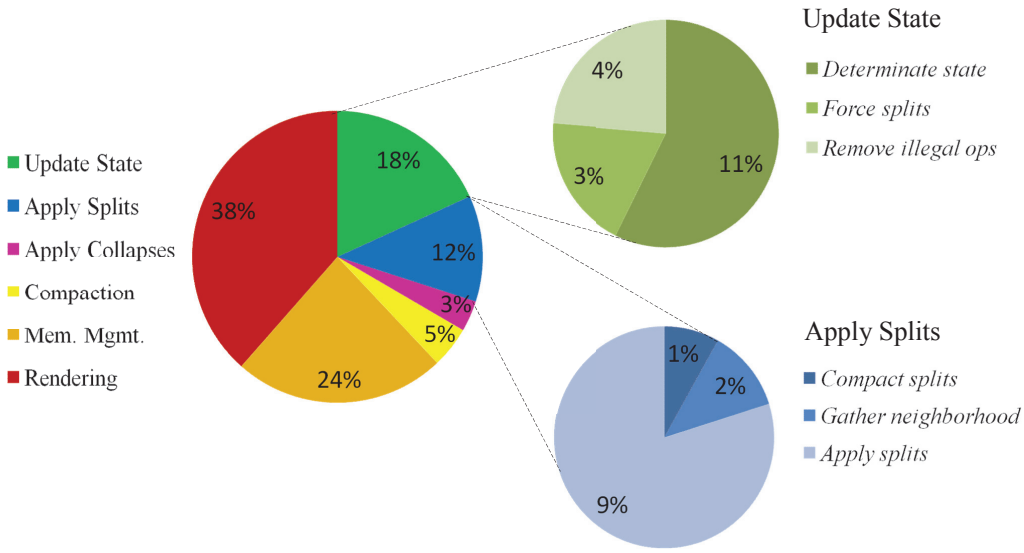

Figure 5.8: Time per frame partition for the serveral steps of the proposed algorithm and for map-/unmap as well as rendering.

instead of several geometry shaders to construct a compact indexed buffer for rendering. In addition, rendering the adapted mesh is also approximately twice as fast using the proposed approach due to the compact vertex buffer. Due to ther stronger neighborhood contraints the proposed method however requires thrice the number of iterations to reach the specified LOD than the method of Hoppe et al. [HSH09]. Since on the other hand each iteration is more than nine times faster, the proposed method still converges in a third of the time. Nevertheless, popping artifacts are still visible during a fast pan over the model which can be observed in the accompanying video. Note that the relatively large amount of time required for memory management could be reduced by using larger blocks during allocation. This would on the other hand increase the amount of memory used up by the dynamic data structures.

Even with most recent graphics drivers approximately 25% of additional rendering time is required for the mapping and unmapping of the index and vertex buffer for access from CUDA. According to the documentation the time for mapping should be insignificant if the device is set to OpenGL interoperability. As I have observed no difference between activating OpenGL interoperability or not, I consider this to be a driver problem and did not include this time in the results. This is also one of the reasons for the rather large share of the memory managemant as allocating a new VBO requires unmapping the old and mapping the new one.

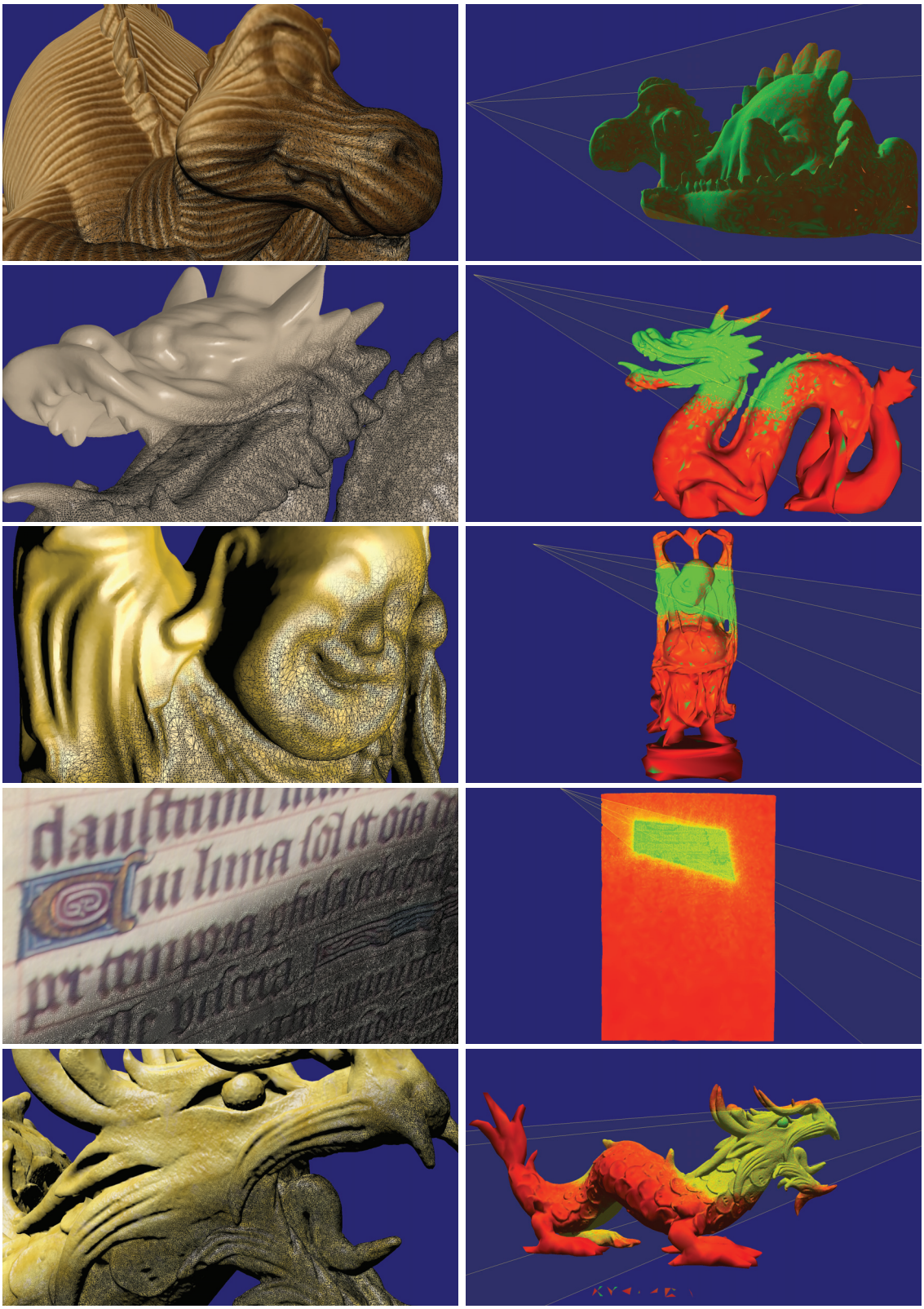

Figure 5.9: Renderings of view-dependently refined meshes. The images on the right show external views with the view frustum in yellow. The color coding depicts the level of detail, where red is low LOD and green high.

5.2.4 Conclusion and Limitations

In this section a compressed progressive mesh representation is proposed. It was specifically developed for parallel refinement on modern graphics hardware. By performing all currently possible adaption operations in parallel, the algorithm only need 1.6 times as long as for rendering of the adapted mesh. In total the algorithm need 2.6 times as long as for rendering alone. This means that the performance is increased as soon as 62% of the vertices are removed by simplification. Due to the view-dependent adaption, this reduction is almost achieved by coarsening the back faces alone. Compared to prevoius parallel view-dependent refinement algorithms the proposed method achieve an almost ten-fold performance improvement. The proposed algorithm even outperforms hierarchical LODs that were considered near-optimal for current graphics hardware by a factor of more than two.

In addition to the improved performance, the proposed method also requires even less graphics memory than the original model stored as indexed face set. For larger models, approximately 30% to 40% are saved on average while other algorithms need more memory than the original model.

One limitation of the proposed algorithm is that despite sorting the triangle into fans to utilize the vertex cache, an additional memory reduction would be possible by using a generalized triangle strip. Another, probably more severe limitation is that some splits are postponed several frames as they are waiting for others to be applied before them. Although this is only problematic for fast panning over the model, a less restrictive dependency scheme would be desirable.

5.3 Parallel View-Dependent Out-of-Core Progressive Meshes[36]

The proposed view-dependent out-of-core refinement algorithm is based on the previously discussed *Parallel View-Dependent Refinement of Compact Progressive Meshes* in-core algorithm (Section 5.2) which is briefly describe before discussing the modifications.

5.3.1 Overview

After building a progressive mesh, a view-dependent reconstruction is generated by performing only those split operations necessary for the current view. During this process the local ordering of operations needs to be preserved. This leads to the dependency rules formulated by Hoppe [Hop97]:

- The ordering of operations applied to a single vertex must be preserved.

36 In proceedings of Vision Modeling and Visualization (VMV2010) [DMG10b].

- A split can only be applied if the next split operation of each neighboring vertex was generated earlier during simplification.
- Edge collapse operations are only legal if the next collapse of each neighboring vertex was created later.

The first dependency rule can be efficiently encoded in a forest of binary trees as shown in Figure 5.10. For each vertex of the base mesh, a binary tree is constructed.

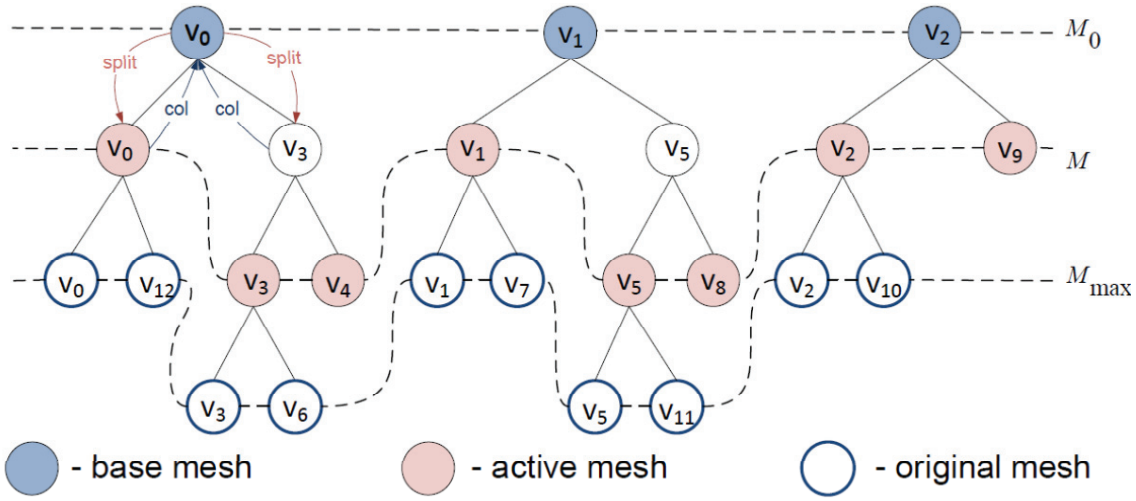

Figure 5.10: Split/collapse operation hierarchy represented as a forest of binary trees.

The operation indices cannot be used to preserve the local ordering because their sequence is not preserved. Therefore the neighborhood dependency is encoded explicitly for each operation. For compact encoding the main idea is to construct consecutive independent sets:

1. Start with base mesh $M_i = M_0$.
2. Store all currently possible operations in level i.
3. Perform all operations of level i on M_i to construct M_{i+1}.
4. Increment i and continue with the second step until no operations are left.

The split level is then stored for each operation and the local ordering is preserved if only the vertex with lowest level in each face is split. Accordingly, only the vertex with the highest level in each face can be collapsed.

A compact data structure is used to store the topology and attribute modifications of each operation. The up to two new faces created by a split operation are defined by two neighbor vertices v_l and v_r. These are stored by first defining an ordering on the vertex neighborhood. For this purpose an operation index i is used and the index of each vertex is either its base mesh index or the operation index plus the number of base mesh vertices.

Then, v_l and v_r are encoded by their rank in the ordered sequence of neighbor vertices. In the example shown in Figure 5.11, the ranks are 0 and 4. In addition, the connectivity modifications are encoded using an ordering of the neighbor triangles. The triangle index is calculated analogously to the vertex index. A bit flag is then set to one for each triangle adjacent to v_u after the split. In Figure 5.11 the modified faces and the resulting bit vector are shown.

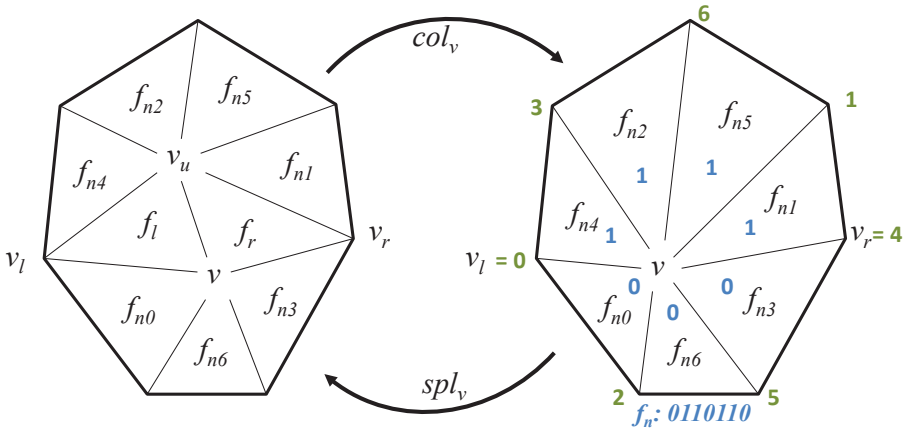

Figure 5.11: Example of topology encoding.

Both, the ranks of v_l and v_r, and the bit flag are encoded with fixed length variables. The valence of the vertices is restricted to 15 and the number of neighbor faces to 16. Thus 8 bits are sufficient for the new faces and 16 bits for the connectivity modifications. Vertices with higher valence are not collapsed during simplification. As the collapse of two neighbor vertices reduces their valence they will be collapsed at a later stage if suitable.

In addition to the topology, the attribute differences (position, normal, texture coordinates etc.) of v and v_u are stored. First, each attribute is scaled such that the variances are confined to the same range. Then the differences are scaled to the range $[-1,1]$ per operation and quantized to n bits using a cubic function. To reduce the quantization error, dummy splits are introduced if necessary. For the refinement criteria, the simplification distance, normal cone angle, and the ratio of geometric to attribut error are stored.

The adaption algorithm additionally maintains a few dynamic data structures. They store the split hierarchy as well as a dynamically updated vertex buffer and index buffer. The main data structures required for rendering are the vertex buffer, which contains the position and attributes of the adapted vertex and the index buffer that contains the connectivity. Both are stored as vertex buffer objects (VBOs) and are therefore separated from all other data. Together they form the indexed face set that is used for rendering. To store the prefix sum, neighborhood and collapse information additional memory is required.

5.3.1.1 Spatial Operation Hierarchy

For out-of-core rendering a spatial hierarchy of split/collapse operations is now build. The advantage over previous approaches is that this way no special boundary constraints between adjacent hierarchy nodes are introduced. The hierarchy serves two purposes: first of all, the operations should be grouped such that those which are likely to be performed simultaneously or successively are stored together. In addition, it is also to be used for occlusion culling in order to coarsen invisible parts of the model. During hierarchy construction it thus needs to be optimized for both purposes.

For occlusion culling, Meißner et al. [MBH*01] proposed a simple heuristic to construct efficient kd-tree hierarchies for triangle meshes using a greedy algorithm. It is based on the idea to minimize the total area of all bounding boxes in a hierarchy with a fixed maximum number of triangles n_{max} per leaf node. Starting from the root node, an optimal partition with respect to an estimated total bounding box area is performed. The area A is estimated as:

$$A \approx A_l \left(1 + \log_2 \left\lceil \frac{n_l}{n_{max}} \right\rceil\right) + A_r \left(1 + \log_2 \left\lceil \frac{n_r}{n_{max}} \right\rceil\right), \tag{5.12}$$

where A_l and A_r are the bounding box areas of left and right child node and n_l and n_r the respective number of triangles. Finding the optimal split is performed by sorting the triangles in x-, y-, and z-direction and then calculating the estimated area for all ordered partitions.

In contrast to a hierarchy for triangles, the operations are not only stored at leaf level but also at inner nodes to reconstruct coarse approximations of the model. When processing a node, the operations need to determined first that are stored in the current node. Then their subtrees are partitioned into the child nodes. Finding the directly stored operations is rather straightforward as those with the highest simplification error are required first. Since it is not efficent to skip a node for two successive refinements of a vertex, all operations which are referenced from the previous node are added first. For the root node, these are the operations of the base mesh. When the operations are stored in the current node their next operations are partitioned into the child nodes. This way a complete operation subtree is stored in a single subtree. Due to storing operations not only at the leaf nodes, the estimated area is slightly different than in the approach of Meißner et al.:

$$A \approx A_l \log_2 \left\lceil \frac{n_l}{n_{max} + 1} \right\rceil + A_r \log_2 \left\lceil \frac{n_r}{n_{max} + 1} \right\rceil, \tag{5.13}$$

where n_* now is the number of operations.

5.3.1.2 Tree Structure

The child relations of the operation tree are encoded by storing the operations relative to the current node. The relative indices are stored as c_l for left and c_r for right child operation. Each operation is accessed using two values: the node index i_n and the local index of the operation in this node i_l. A unique global index i which is required later can be computed as:

$$i = n_{max} i_n + i_l \tag{5.14}$$

The node and local index of the child nodes are calculated as follows:

$$i_{n,l/r} = \begin{cases} c_{l,r} < n_{max} & : \ i_n \\ n_{max} \leq c_{l/r} < 2n_{max} & : \ left\_child(i_n) \\ c_{l,r} \geq 2n_{max} & : \ right\_child(i_n) \\ \text{else} & : \ \text{no operation} \end{cases} \tag{5.15}$$

$$i_{l,l/r} = c_{l/r} \mod n_{max}, \tag{5.16}$$

where i_n is the node of the current operation.

To lessen the constraint of 256 split levels as in the in-core case, 10 bits are used instead of only a single byte for the level. Together with 11 bits for each of the child indices, the ordering and dependency are encoded in four bytes per operation. Using the index coding described above, up to 682 operations can be stored per node.

5.3.1.3 Data Structures

In contrast to the *Parallel View-Dependent Refinement of Compact Progressive Meshes* in-core algorithm (Section 5.2) only the currently required parts of the static data are stored in graphics memory. By using these data structures, the view-dependent mesh can be refined and rendered in real-time with minimal memory requirements. Table 5.6 shows the static and dynamic data structures required to maintain and update the vertex and index buffers in detail. Differences to the in-core algorithm are marked in blue. Since the complete algorithm runs on the GPU, all the relevant data is stored in graphics memory and the main memory consumption is minimal. Besides temporary memory for loading, only the mapping of nodes to memory positions and file offsets required for out-of-core management are stored in main memory.

As the operations are referenced using a node and a local ID, these are stored instead of the next split or collapse operation index. To apply the operations the operation hierarchy also need to stored. To determine which nodes are loaded, pointers of the operation nodes are additionally stored in device memory. For occlusion culling the visibility of the nodes is additionally stored. In summary, the out-of-core data management requires 1.875 additional bytes for each operation, 4 bytes for each active vertex and 13 bytes for

buffers	elements	memory (bytes)
static structures		
nodes	nodes	$8o$
	offset	$4o$
	visibility	$1o$
operations	tree structure	$3n$
	dependency	$1n$
	ref. criteria	$3n$
	topology	$3n$
	quant. delta	$2kn$
	delta scale	$2n$
dynamic structures		
active faces	index VBO	$24m$
	triangle ID	$8m$
active vertices	vertex VBO	$4km$
	vertex ID	$4m$
	node ID	$4m$
	local ID	$2m$
	next collapse	$4m$
	v_{state}	$1m$
collapse tree	node ID	$4m$
	local ID	$2m$
	prev. collapse	$4m$
	v_u	$4m$
temporary	prefix sum	$24m$
	neighb. size	$1m$
	neighb. index	$16m$
total		$13o + (12 + 2k)n$ $+(102 + 4k)m$

Table 5.6: Elements of the data structure. o, k, n, and m are the number of operation nodes, attributes, operations in graphics memory, and vertices of the adapted mesh.

each node. As each node stores many operations, their number is rather low. In total $1.875n + 4m + 13o$ bytes additional memory is require for the out-of-core data management compared to *Parallel View-Dependent Refinement of Compact Progressive Meshes* in-core algorithm (Section 5.2). On the other hand, the memory requirements for the static data are drastically reduced as only a subset of the operations is kept in graphics memory.

5.3.2 Runtime Algorithm

For the dynamic data structures the same block based memory management as in *Parallel View-Dependent Refinement of Compact Progressive Meshes* in-core algorithm (Section 5.2) is used, but additionally introduce an out-of-core management for the static data structures.

5.3.2.1 Out-of-Core Memory Management

The operations and attributes are subdivided into the nodes described above and stored in a large file. Then only the currently necessary nodes are loaded into graphics memory. A node is required when at least one of the active vertices or collapses has a reference to it. A split operation can create vertices that reference nodes not available in graphics memory. In this case, this node is loaded from disk into graphics memory. Before loading the currently required nodes all inactive ones are removed first.

Since accessing the hard disk is a severe bottleneck, loading nodes into main memory is preformed in a second thread. As soon as the data is available in main memory, the rendering thread can copy it into graphics memory. To prevent strong frame rate fluctuations only a maximum number of l_{max} nodes per frame is loaded, because the memory transfer to the graphics card is relatively expensive. This approach slightly slows down the adaption, but guarantees stable frame rates. If the number of required nodes exceeds l_{max} the ones that were requested the highest number of consecutive frames are loaded. This has the advantage that the model is uniformly adapted and no LOD starvation can occur.

The operation memory is shared for all progressive models of the scene and a maximum number of nodes o_{max} kept in graphics memory is specified. If the number of totally required nodes exceeds o_{max}, a global level of detail scaling factor is used to coarsen all models until enough memory is available. If later more free node are available the factor is gradually reduced again.

5.3.2.2 Parallel Adaption Algorithm

In order to optimally exploit the parallel architecture of the GPU, the adaption algorithm is subdivided into several consecutive steps. Each step is then performed in parallel. The algorithm is based on 4 bit-states to encode possible and necessary operations and 2 temporary states for collapsed and removed vertices. It is composed of the five following main stages, where the first three steps are similar to the in-core variant.

In the first step the state of the vertices is updated, where the refinement criteria determine which vertices need to be split and which can be collapsed. In addition to the original refinement criteria – i.e. view-frustum culling, back-face culling and projected error – occlusion culling is also used to further reduce the number of active vertices. The occlusion culling is based on the operation nodes whose visibility is determined after rendering.

If the next split or collapse of a vertex is in an occluded node, the vertex itself is marked as invisible. Visible (green) and invisible (red) blocks of a model are shown in Figure 5.12. After checking if a vertex is occluded, the original refinement criteria are tested to determine which operations need to be performed. When the necessary operations are determined, the splits of neighbor vertices needed to force due to the face dependencies

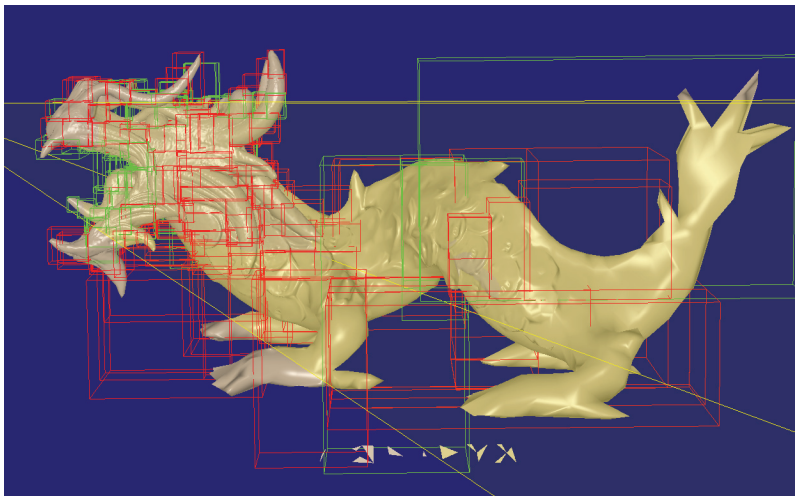

Figure 5.12: Red blocks - out of view frustum or occluded. Green blocks - visible.

and all impossible operations removed. Since not all static data is available in graphics memory, the algorithm need to check whether the required operation and the operations of its neighbor vertices are available. If this is not the case, the operation need to delay.

In a second step all remaining operations are applied. For this purpose the neighborhood of the split vertices is collected first and then the split operations are performed. Applying a split operation includes generating the new vertex v_u, moving the vertex v and creating two new faces f_l and f_r. After the split operations the collapse operations are performed. For each collapse vertex v the corresponding vertex v_u is removed, the faces are updated and degenerate ones are removed.

The third step of the adaption is the compaction of buffers where elements have been removed. These buffers are the active vertices (including the vertex VBO), active faces (with the index VBO), and collapse operations. To improve rendering performance, the index VBO is periodically sorted to exploit vertex caching.

Finally the algorithm determine which of the nodes are occluded to use this information for the next frame. For this purpose hardware occlusion queries [CCG*01] are used and render the bounding boxes of the nodes. The query is performed after rendering the complete scene and the result is fetched before the next parallel adaption. This way the visibility is lagging one frame behind which is not problematic as it is only used for LOD selection, but not for rendering. Regardless of the query results the complete adapted mesh is always rendered. This approach is very efficient because the number of the active nodes is small compared to the number of triangles, only asynchronous queries are used, and only require a single switch between rendering and occlusion queries per frame.

5.3.2.3 Prefetching

When the user moves through the screen, the visible portions and the required LOD of the object change. This results in a continuous change of the nodes currently required in device memory. Due to the high latency of the hard disk, loading out-of-core data results in a visible delay of the mesh refinement. This problem is solved with a two-step prefetching algorithm: First, not only the currently required nodes are loaded, but also their direct children if enough space is available. In addition, the current camera movement is extrapolated and the view frustum is enlarged to contain all frusta of the subsequent 10 frames. This results in a dynamic adaption of the view frustum, which means that the algorithm pre-refine parts of the model before they become visible.

Note that the prefetching algorithm is memory sensitive: it only works if enough memory is available on the device. This is achieved by assigning a low priority to all prefetching candidates, so these nodes are only processed after all other ones are loaded.

5.3.3 Results

The test system is built of a 2.4 GHz Intel Core2 Duo CPU with 2 GB of DDR2 main memory, 16 lanes PCIe slot, and a GeForce GTX 480. The OpenGL API is used for rendering and CUDA to implement the parallel algorithm. The out-of-core progressive meshes are stored on a Seagate SATAII hard disk with 7200 rpm and 32 MB cache. The access time is 8.5 ms and approximately 100 MB can be read per second. The resolution of 1920×1080 is used in all experiments. Table 5.7 gives an overview of the progressive meshes that are in the experiments used.

model	v_0	f_0	*# ops.*	*# dummy ops.*	lvl.	nodes
Asian Dragon	40	19	3,612,383	2968 (0.08%)	213	5507
David	659	1416	3,615,968	2529 (0.07%)	276	5536
Statuette	16	40	5,014,234	14254 (0.28%)	212	7678
Lucy	15	29	14,040,204	24681 (0.17%)	268	21421
Sponza scene	730	1504	26,282,789	44432 (1.69%)	276	40142

Table 5.7: Progressive meshes used in the experiments, number of base mesh vertices, base mesh faces, operations, added dummy split operations, maximum split level and nodes.

All models use position and normal as vertex attributes only (i.e. $k = 6$ attributes). The resulting file sizes – compared to an indexed face set – are listed in Table 5.8. The file size reduction is almost identical for all models and is approximately 50% due to identical number of attributes. Note that HLODs and Quick-VDR both need 50% to 80% more disk storage than an indexed face set.

Table 5.9 shows the number of rendered faces, the total rendering time, and the memory consumption for the views shown in Figure 5.13 and the Sponza scene in the accompanied

model	v_{max}	f_{max}	*mem. IFS*	*mem. PM*
Asian Dragon	3,609,455	7,218,906	165.2MB	82.9MB (50.2%)
David	3,614,098	7,227,031	165.4MB	83.0MB (50.2%)
Statuette	4,999,996	10,000,000	228.8MB	115.0MB (50.2%)
Lucy	14,027,872	28,055,742	642.2MB	322.1MB (50.2%)
Sponza scene	26,251,421	52,501,679	1201.6MB	603.0MB (50.2%)

Table 5.8: Number of original mesh vertices and faces and comparison of the static data (PM) to an indexed face set (IFS).

model	*rendered # faces*	*memory (MB)*	*total frame time (ms)*
Asian Dragon	853,667 (11.8%)	112.07 (67.9%)	11.0 (89.7%)
David	916,044 (12.6%)	115.48 (69.8%)	11.8 (87.0%)
Statuette	1,450,698 (14.5%)	144.05 (62.9%)	18.7 (88.0%)
Lucy	2,125,849 (7.6%)	224.59 (34.9%)	24.9 (20.3%)
Sponza scene	2,116,626 (4.0%)	220.65 (18.3%)	33.2 (19.8%)

Table 5.9: Memory consumption and total rendering time of the different models. The ratio compared to rendering an indexed face set of the original model is shown in parenthesis.

video, where the numbers are taken from the most complex frame. For the Asian Dragon, David and Statuette 4096 nodes (64.0 MB) are used. For the Lucy and Sponza scene 6144 nodes (95.9 MB) are used, whereas each node contains 682 operations. During rendering, the dynamic data structures consume additional memory. For all models, the total amount of graphics memory nevertheless stays below that of an indexed face set. The savings to an IFS are about 30% for the medium sized and up to 82% for larger models. This allows to safe up to 1 GB of graphics memory for the sponza scene and over 80% of the rendering time. Figure 5.13 also shows the coarsening of the back faces, faces outside the view frustum and occluded ones.

Table 5.10 shows the number of rendered faces, the total rendering time and the memory consumption of the in-core algorithm for identical views. Comparisons to David, Lucy and Sponza scene are not possible, because the split level is too high. With the occlusion-culling the number of active faces is significantly (over 50%) reduced. Additionally, the algorithm need about 44% less memory and up to 29% less rendering time for identical views.

model	*rendered # faces*	*memory (MB)*	*total frame time (ms)*
Asian Dragon	1,740,953 (+103.9%)	174.02 (+55.3%)	15.4 (+40.0%)
Statuette	2,536,238 (+74.8%)	246.25 (+70.9%)	21.1 (+12.9%)

Table 5.10: Memory consumption and total rendering time of the *Parallel View-Dependent Refinement of Compact Progressive Meshes* in-core algorithm (Section 5.2). The ratio compared to the values in the Table 5.9 for identical views.

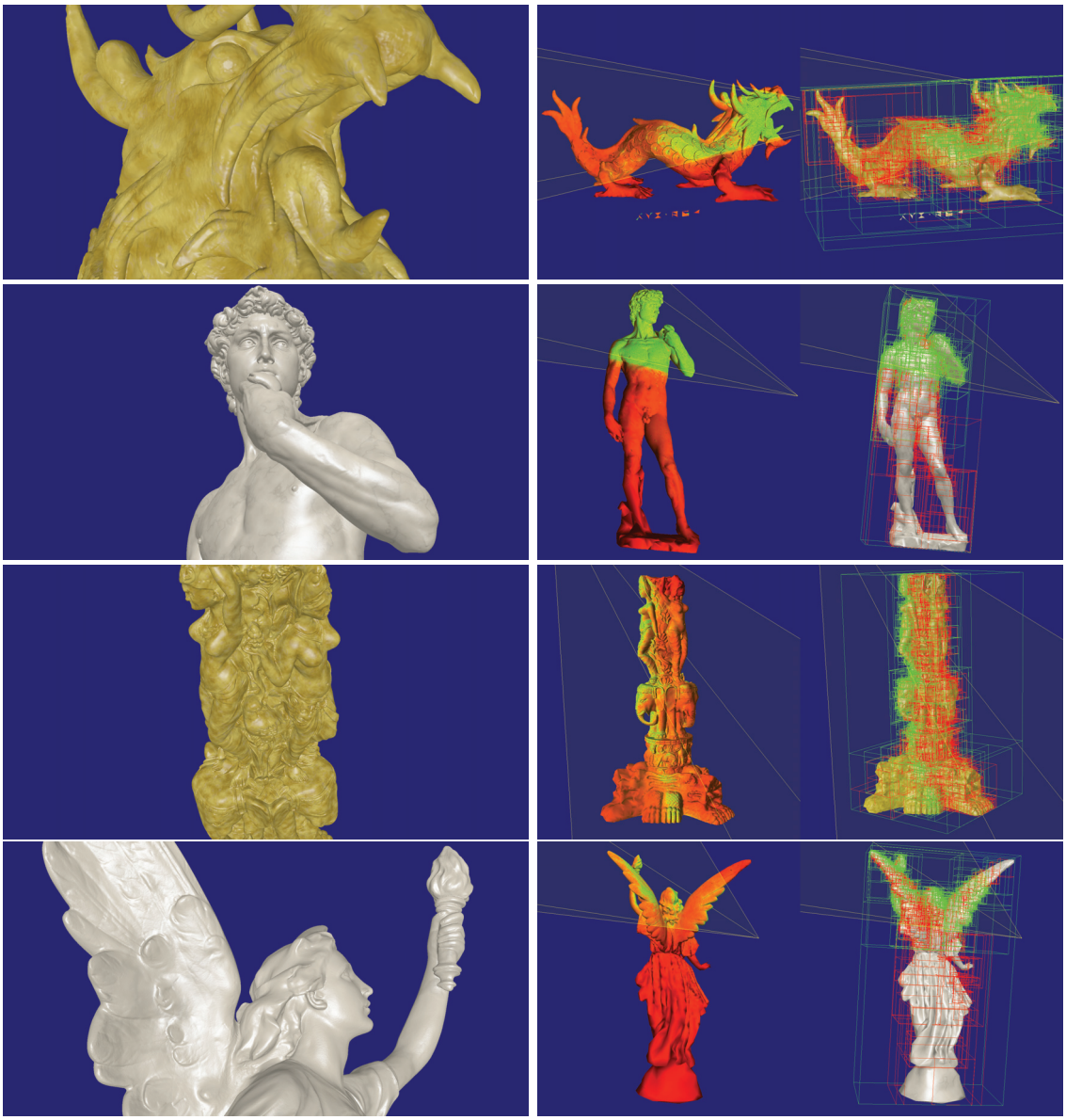

Figure 5.13: Renderings of view-dependently refined meshes. The images on the left show the models as rendered from the point of view. In the middle external views with the view frustum (yellow) are shown. The color coding depicts the level of detail, where red is low LOD and green high. The image on the right shows the nodes used for occlusion culling. Occluded nodes are shown in red and visible ones in green.

Compared to static hierarchical LODs (HLODs) [GBBK04] using the same error measure, the number of primitives is reduced by a factor of 3 to 5 and the frame rate improves by a factor between two and three. On the used test system, for the David model and identical view, the HLOD rendering requires approximately 4.1 M faces, 175 MB graphics memory and 56 ms (17.8 fps) for rendering. When the LOD switches, the frame rate

can even drop below 10 fps. This is due to the fact that the LODs can only be selected based on the viewing distance as only a single mesh is stored per node. The performance of Quick-VDR [YSGM04] is only slightly better than static HLODs while the number of triangles is approximately halved. On the used test system, for the David model and identical view, Quick-VDR requires approximately 2.1 M faces, 200 MB graphics memory, 600 MB main memory and achieves 25-30 fps. Additionally, the adaption of the Quick-VBR is very slow. It achieves approximately 60 k operations per second on my test system whereas the proposed approach achieves over 1.2 M operations per second.

Figure 5.14 and 5.15 shows the adaption and rendering time, the memory consumption, and the number of active faces for a pre-recorded movement for the Asian Dragon. The frame time and consumed graphics memory is always significantly lower than required by the in-core algorithm, because the number of active faces is reduced with the occlusion-culling significantly (up to 60%). In addition to the reduced frame time up to 50% less memory is required.

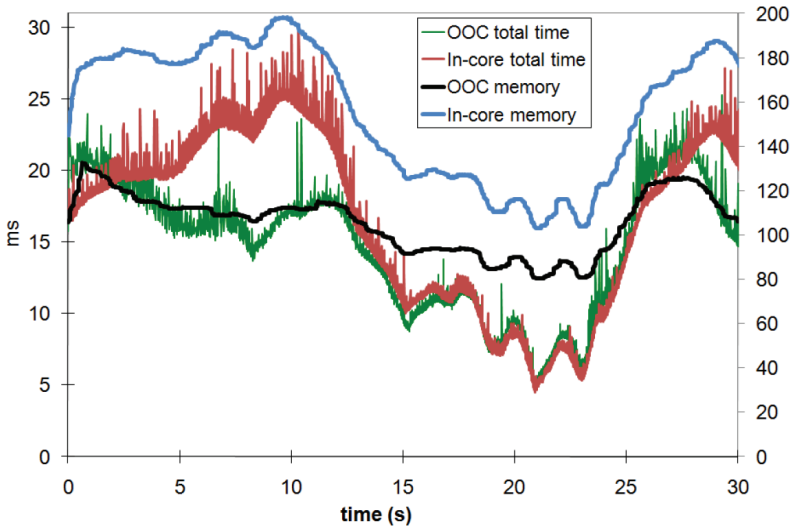

Figure 5.14: Comparison of timings and memory consumption of the proposed approach (OOC) with in-core algorithm for the Asian Dragon with a pre-recorded camera path.

Figure 5.16 and 5.17 shows the adaption and rendering time together with the memory consumption and the number of active faces for a pre-recorded movement through the Sponza scene. The consumed graphics memory is always below 220 MB. The frame rate is constantly about 50 frames per second, with some drops down to 40 fps. The number of faces during the walk through slightly exceeds 2 millions in some situations. The proposed approach quickly reacts to changes of the view direction with fast adaption of the scene complexity.

Since the rendering performance is identical to rendering a static model with the same number of triangles, the proposed method needs approximately five times as long as rendering a static mesh. Considering that the algorithm already cut down the vertices

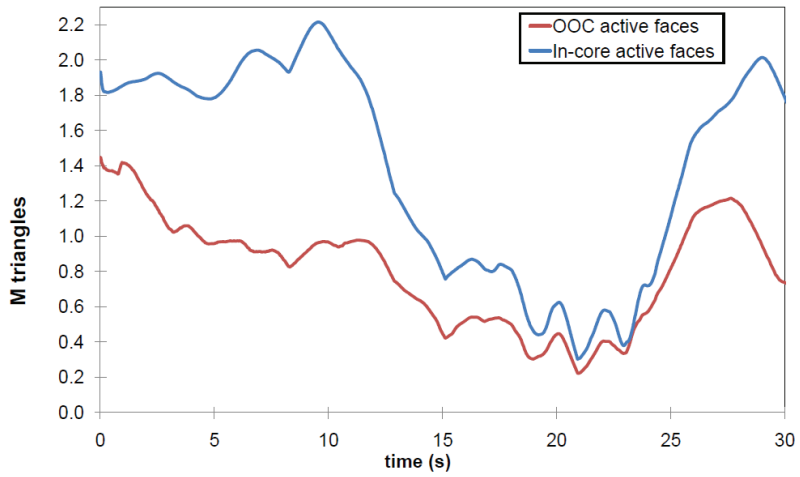

Figure 5.15: Comparison of the number of active faces of the proposed approach (OOC) with in-core algorithm for the Asian Dragon with a pre-recorded camera path.

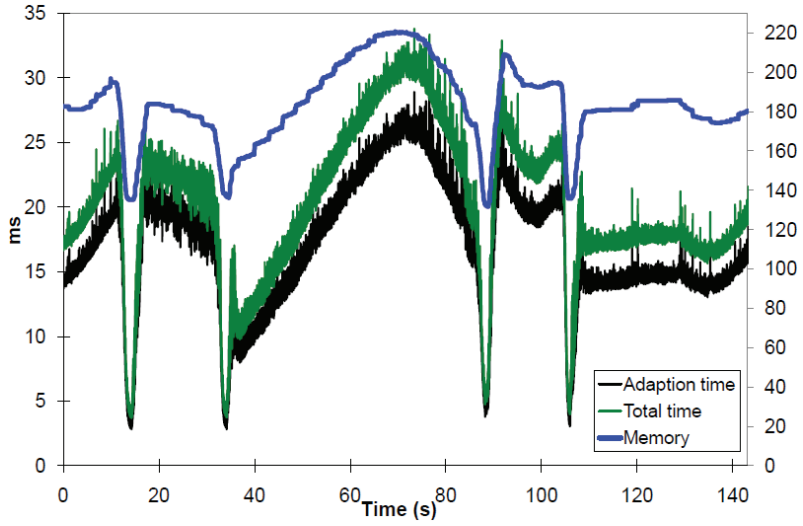

Figure 5.16: Timings and memory consumption for the Sponza scene with a pre-recorded camera path.

significantly due to the simplification of back-faces, faces outside the view frustum and occluded faces. I can conclude that the proposed method will almost always be faster than rendering an indexed face set of the original model. While this even holds for rather coarse models, the performance gain increases with the complexity of the original mesh. Due to the time required for the pixel shaders, the speedup is of course not linear with the reduction.

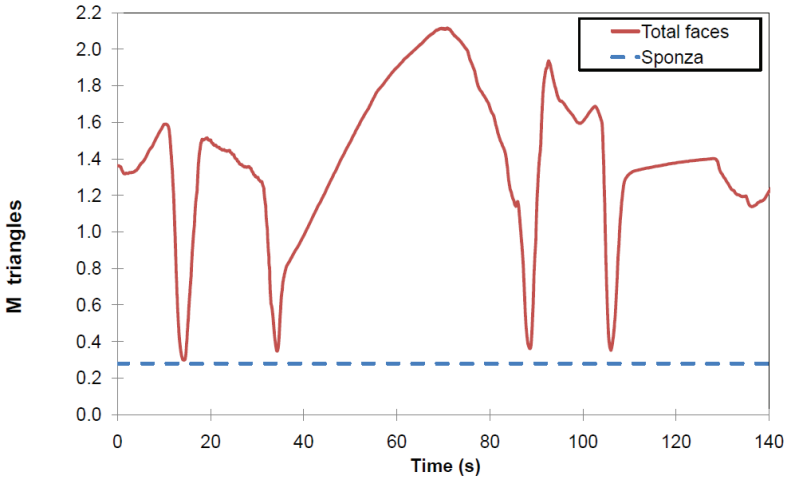

Figure 5.17: The number of faces for the Sponza scene with a pre-recorded camera path.

5.3.4 Conclusion and Limitations

In this section an out-of-core progressive mesh representation was proposed. It was specifically developed for parallel refinement on modern graphics hardware. In this section proposed algorithm extends the *Parallel View-Dependent Refinement of Compact Progressive Meshes* in-core algorithm (Section 5.2) with out-of-core data management and is the first out-of-core approach that is completely based on view-dependent progressive meshes. The dependency and operation coding are modified for large models and the dynamic data structures, static data structures, and first three steps of the algorithm are extended for out-of-core data management. In addition, a spatial hierarchy for the operations that is also used for occlusion culling was proposed.

The proposed algorithm remove more than two thirds of the vertices by view-dependent simplification compared to HLODs. Due to the much finer granularity, this reduction is practically always surpassed. Compared to HLODs the frame rate is more than double. In addition, the frame rate drop when changing the LOD is insignificant whereas for HLODs the frame rate can drop by a factor of more than two.

One limitation of the proposed algorithm is that an additional memory reduction would be possible by using a generalized triangle strip. Another, probably more severe limitation is that some splits are postponed several frames as they are waiting for others to be applied before them. Although this is only problematic for fast panning over the model, a less restrictive dependency scheme would be desirable.

5.4 Dependency Free Parallel Progressive Meshes[37]

The proposed compressed progressive mesh data structure is based on Hoppe's original progressive mesh algorithm [Hop96]. The progressive mesh is generated by simplifying the original mesh with a sequence of collapse operations until no faces are left. The original mesh can then be reconstructed by applying the corresponding split operations in reverse order.

5.4.1 Overview

Figure 5.18 shows an edge collapse operation col_v which removes the vertex v_u and modifies v_t to v. The adjacent faces f_l and f_r of v_t and v_u degenerate and are removed from the mesh. The corresponding vertex split spl_v inverts this operation. Accordingly the faces f_l and f_r are generated when the vertex v is split into v_t and v_u. In addition, some of the faces adjacent to v become adjacent to the new vertex v_u, the others remain connected to v_t.

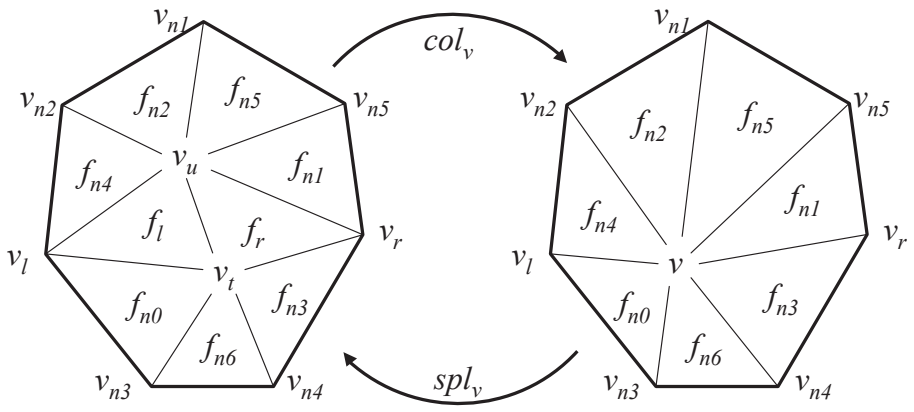

Figure 5.18: Edge collapse and vertex split operation.

After building a progressive mesh, a view-dependent reconstruction can be generated by performing only those split operations necessary for the current view. Performing a local adaption requires a random access data structure that allows to locally perform the operations. While the operations are already local by definition, the method of encoding the connectivity strongly influences the degree of locality. The main idea of the proposed data structure is to store the connectivity changes in the triangles instead of storing it within the operation. This way, the connectivity of each face can be updated without considering its current neighborhood.

37 In Computer Graphics Forum 2012 [DG12].

5.4.1.1 Neighborhood Dependencies

Originally, Hoppe [Hop96] explicitly encoded the vertex indices of v_l and v_r, and the indices of the faces adjacent to v_u. Xia et al. [XV96] optimized the data structures by encoding the vertex and face indices relative to the neighborhood of the split operation. The memory consumption can be drastically reduced this way since v_l and v_r can be encoded in a few bits. This encoding was previously used in the algorithms proposed in the section 5.2 and 5.3 for parallel view-dependent refinement. While view-dependent adaption is possible, the local dependencies require that v_l, v_r, and $v_{n1} - v_{n5}$ exist when performing a split operation of v (see Figure 5.18). Hoppe [Hop97] proposed a slightly different approach that does not require v_l and v_r to be present, but nevertheless forces splitting of their ancestors afterwards to prevent foldovers. These of course cannot be encoded as compactly as in the previous approach. The faces adjacent to v_u are then found by traversing the edges in clockwise order from v_l to v_r. Hu et al. [HSH09] later used a modification of this technique for parallel refinement. In both cases, the simplification constructs a forest of binary trees (Figure 5.19). The neighborhood dependencies (dotted lines) are either encoded explicitly, or implicitly by using a special numbering of the vertices.

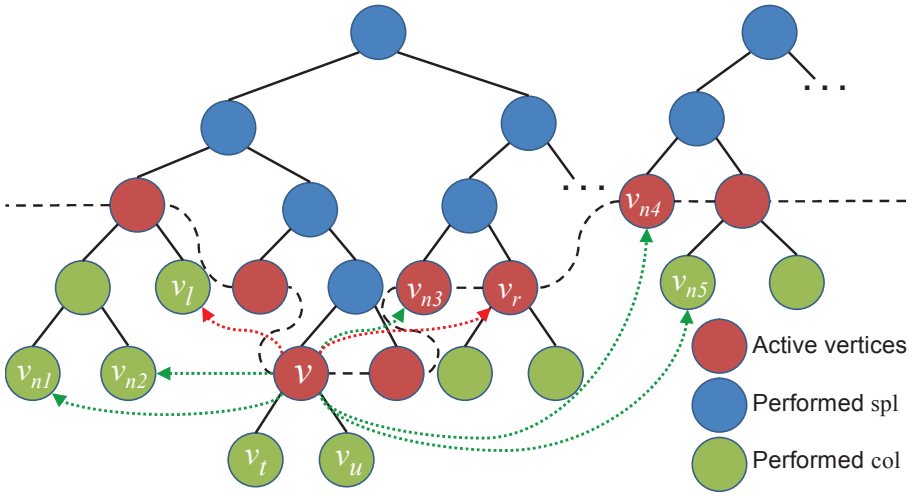

Figure 5.19: Vertex hierarchy represented as a forest of binary trees with full (green) and reduced (red) neighborhood dependencies.

The drawback of these approaches is that when vertex v needs to be split, it often needs to wait for neighboring splits. Figure 5.20 shows such a case. The vertices v_1 and v_3 need to split before v_4. In addition, v_2 also needs to split if all neighboring vertices are required. As splitting v_3 requires v_{2u} and v_2 requires v_{1u}, the splits can only be performed sequentially. While this is unproblematic in a sequential refinement algorithm, a parallel algorithm needs four adaption passes. By removing all neighborhood dependencies, the proposed method can split all four vertices in parallel.

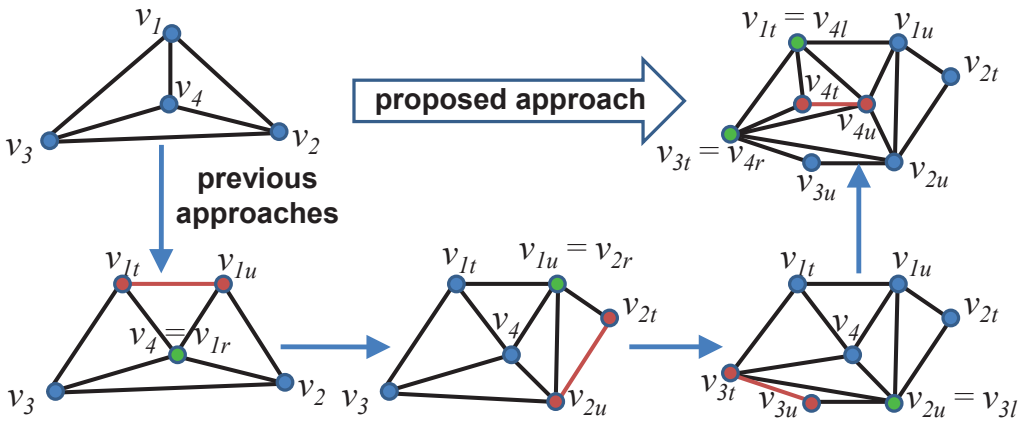

Figure 5.20: Dependent split operations. Each arrow denotes a parallel adaption step.

5.4.1.2 Split Operations

While previous approaches require at least v_l and v_r to exist, the proposed algorithm remove this constraint and only require v to exist. This allows to perform splits of all currently active vertices completely in parallel. In the example above (Figure 5.20), all four vertices can be split within a single adaption pass. The proposed algorithm achieve this by encoding all possible topology modifications within the vertex indices of the faces. The faces are then stored along with the split where they are generated. To support non-manifold meshes, the proposed algorithm first store the number of generated faces and then the faces themselves. The faces are encoded by storing their vertex indices $\text{FVID}_{0..2}$ for the finest resolution. The current indices can then be found by searching those vertices into which the final vertices are collapsed. Thus the vertices need to be numbered such that the proposed algorithm can efficiently find the currently active vertex into which the vertex with FVID_i is collapsed. Figure 5.21 shows this numbering of the vertices. The leaf nodes of the binary tree forest are simply numbered from left to right. Then the collapsed vertex v receives the FVID of its left child v_t which is the smaller one. The resulting encoding of the faces is shown on the right side of the figure for a simple model. If a face is now decoded, when the split operation is performed, the current vertices are either those with the same FVID, or the one with the greatest FVID smaller FVID_i. In the example, the algorithm need to find the active vertex for the FVID 1 when performing split 1. The currently active vertex with the greatest FVID smaller than 1 is vertex 0 which is the collapse target of vertex 1.

In theory this can lead to foldovers as the generated triangles can be flipped if v_l or v_r change their position. This can however not happen if the simplification errors of collapsing v_l and v_r are at least that of collapsing v. Note, since the edge collapses are generated with increasing error, this is automatically handled during simplification. The local monotonicity is enforced by tracking the simplification error of the adjacent vertices

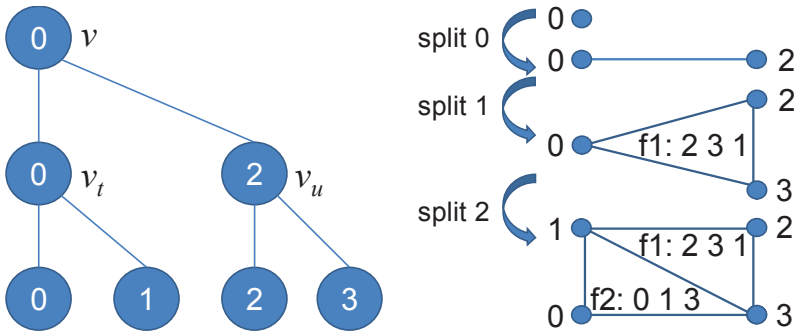

Figure 5.21: Computation of the final vertex IDs and encoding of the generated faces.

during simplification and using the maximum of all neighbor's errors as the actual error. Now the proposed algorithm only need to make sure that a vertex is split if any adjacent triangle is visible and the simplification error exceeds the screen space threshold to prevent visible foldovers. This would be the case if the model was adapted to a constant error, leading to the same sequence of operations as generated during simplification. Although the algorithm do not explicitly force splitting of v_l and v_r, I did not notice any visible foldovers in the experiments as the error is smoothly changing over the mesh anyways. In addition, each split operation needs to encode the refinement criteria for LOD selection, the vertex attributes of v_t and v_u, and the references to the splits of v_t and v_u. Finally, the operations are compressed using arithmetic coding to reduce the memory consumption of the progressive mesh. In contrast to previous approaches the compression is performed independently for each operation to retain random access.

5.4.1.3 GPU Adaption

The adaption algorithm is subdivided into several consecutive steps to implement the refinements on massively parallel hardware. The partitioning is required for thread synchronisation while each step can be processed completely in parallel. First, each vertex is classified to be split, kept, or collapsed. Then the necessary operations are performed on the adapted mesh. This mesh is then used as input for the next frame to exploit temporal coherence.

5.4.2 Data Structure

Table 5.11 gives an overview of the complete split operation data structure. The proposed algorithm use several view-dependent refinement criteria to determine whether a vertex needs to be split or can be collapsed. It can be collapsed if it is either outside of the view frustum or all adjacent triangles are back-facing. At runtime only the normal of the adapted vertex is available. The algorithm thus encode the maximum angular deviation α from the normal of the simplified vertex forming a normal cone [Hop97]. Since each

group	element	memory (bytes)
connectivity	v_u (FVID)	4
	number of faces	1
	faces (FVID)	$12f$
refinement criteria	α (normal cone angle)	4
	ϵ_g (geometric error)	4
	ϵ_a (attribute error)	4
attributes	Δv_t	$4k$
	Δv_u	$4k$
binary tree forest	children present	1
	child pointer	4

Table 5.11: Elements and size of the uncompressed split operation, where f is the number of generated faces and k the number of vertex attributes.

vertex of the adapted mesh can be adjacent to triangles on different levels-of-detail the algorithm need to consider the normals of all possibly adjacent faces. To prevent the computation of trigonometric functions at runtime, $\sin\alpha$ is stored. A vertex needs to be split if it is visible and the simplification error exceeds some pre-defined limit in screen space. Instead of directly using the quadric error for the LOD selection the algorithm compute the geometric attribute error [GBBK04] after simplification to improve the visual quality. The simplification error is comprised of a geometric error ϵ_g and an attribute error ϵ_a. While the attribute error is independent of the view direction $\mathbf{d}$, the geometric error originates from a displacement in normal direction $\mathbf{n}$. As in most previous approaches, the proposed algorithm do not directly store the attributes of v_t and v_u, but only the differences $\Delta v_{t/u}$ to the attributes of v. This has the advantage that the algorithm can reconstruct the attributes of v from those of v_t and the data stored in the split operation. For the binary tree forest the algorithm first encode whether v_t and v_u are further split. Then pointers to their operations are stored as address offset to the end of the current operation. The algorithm do not need to store an offset for v_t since that operation starts directly after the current one. The proposed algorithm also do not need to store an offset for the next operation of v_u if only a single child is present.

5.4.2.1 Out-of-Core Hierarchy

The algorithm additionally build a bounding volume (BV) hierarchy over the split/collapse operations for out-of-core rendering. The hierarchy serves two purposes: first, the operations are grouped such that those which are likely to be performed simultaneously or successively are stored together. And second, it should be used for occlusion culling in order to coarsen invisible parts of the model. During hierarchy construction it thus needs to be optimized for both purposes. Meißner et al. [MBH*01] proposed a simple heuristic to construct efficient kd-tree hierarchies of triangle meshes for occlusion culling using a

greedy algorithm. In the proposed algorithm this approach was adapted to a bounding volume hierarchy of variable size split operations. Note, that storing operations only is different than the hierarchy used in Quick VDR, where each node contains a complete progressive meshes. The algorithm do not only store operations at leaf level but also at inner volumes to reconstruct coarse approximations of the model. When processing a BV, the algorithm first need to determine the operations that are stored in it. Then the operation subtrees are partitioned into the child nodes. Finding the directly stored operations is straightforward as those with the highest simplification error are required first. After storing the operations in the current node their operation subtrees are partitioned into the child nodes. This way a complete operation subtree is stored in a single BV hierarchy subtree. Due to storing operations not only at the leaf volumes, the estimated subtree area is slightly modified:

$$A \approx A_l \log_2 \left\lceil \frac{s_l}{s_{max}+1} \right\rceil + A_r \log_2 \left\lceil \frac{s_r}{s_{max}+1} \right\rceil, \tag{5.17}$$

where A_l and A_r are the bounding box areas of left and right child node and $s_{l/r}$ the size of the operations in bytes.

5.4.2.2 Operation Encoding

The proposed algorithm use arithmetic compression [Sai04] (Section 2.1.4) to store the operations in graphics memory. Due to this optimal entropy coding, the key to achieve a high compression rate, is to reduce the entropy. Therefore, the rest of this section describes how the algorithm encode the data with low entropy. As each operation needs to be decoded independently, the proposed algorithm use common probability tables but encode each operation in a separate byte stream. This way the algorithm only need the starting address to decode an operation. As the compression changes the length of the data and thus the starting address offsets, the algorithm need to perform a bottom up compression of the operations. With the sequential ordering of operations, this leads to compressing them in reverse order. In addition, the algorithm can only determine the symbol probabilities after compression. Therefore, the algorithm start with probability tables that are constructed with zero offsets and the re-compress the data with the correct probabilities afterwards. The overall compression thus works as follows:

1. Compute uncompressed operations with zero address offsets.
2. Compute the probability tables.
3. Compress the operations in reverse order, computing the correct address offsets.
4. Re-compute the probability tables and re-compress.

The compression rate of arithmetic coding depends on the entropy of the data. Most of the data form zero centered normal distributions that can be compressed quite well. As some of these distibutions contain only absolute values the others are remapped to positive numbers. The algorithm use the following mapping to maintain a normal distribution:

$$u = \begin{cases} v \geq 0 & : \quad 2v \\ v < 0 & : \quad 2|v| - 1, \end{cases} \tag{5.18}$$

where v is a variable from a signed distribution and the u's form a positive distribution.

Arithmetic coding independently processes single bits or bytes to restrict the probability table to a reasonable size. Unfortunately the progressive mesh data do not fit into single bytes but often require 32 or even 48 bits in the out-of-core case. The proposed algorithm therefore use a context based arithmetic compression. A separate probability table is used depending on the byte significance. If a preceding byte of the currently encoded value is non-zero, the probabilities drastically change. In this case an additional table is used to encode the successive bytes (see Figure 5.22).

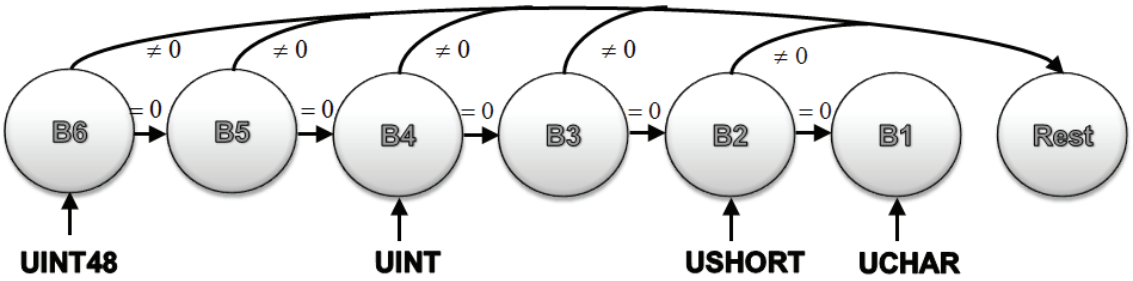

Figure 5.22: To decode 6 byte numbers (e.g. uint combined with ushort) the algorithm begins at table B6, for 4 byte numbers (e.g. uint) at table B4, for 2 byte numbers (e.g. ushort) at table B2 and for 1 byte numbers (e.g. uchar) at table B1. If a preceding byte of the currently encoded value is non-zero, the table Rest is used.

The algorithm perform a bottom up coding of the operation tree compressing the operations from end to start. As the relative symbol frequencies have changed, the algorithm also rebuild the probability table and restart compression. The spatial hierarchy is also rebuilt after each compression run as the operations might exceed the maximum node size. The proposed algorithm iteratively reduce the maximum node size when building the hierarchy and store the model as soon as all nodes are small enough after compression.

Successive Operations

The possibilities of split operations for v_t and v_u are sorted by descending probability and the algorithm store: 0 if none are present, 1 for both, and 2 and 3 for v_t and v_u only. As mentioned above, the proposed algorithm only need to store the address of v_u since the operation of v_t directly starts after the one of v. The address offset o_{addr} for v_u can be estimated as $s_{avg}(v_u - v - 1)$ if the algorithm know the average operation size s_{avg}.

Then only the difference to the estimation is stored. In the out-of-core case the operation address is split into a BV index i_n and the local address depending on the index $addr_l$ inside the hierarchy node due to the partitioning of operations. Using a node size of up to 2^{16} bytes, the combined offset o_{ooc} is stored as:

$$o_{ooc} = \begin{cases} i_n = n & : \quad o_{addr} \\ i_n \neq n & : \quad 2^{16}(i_n - n) + addr_l, \end{cases} \tag{5.19}$$

where o_{addr} is the offset inside the node and n the current BV node. In contrast to the in-core case, o_{ooc} of v_t can be non-zero and needs to be stored as well. Additionally, the offsets cannot be estimated any more since the proposed algorithm do not know how many subtree operations are contained in the current node. Figure 5.23 compares the operation ordering of both cases.

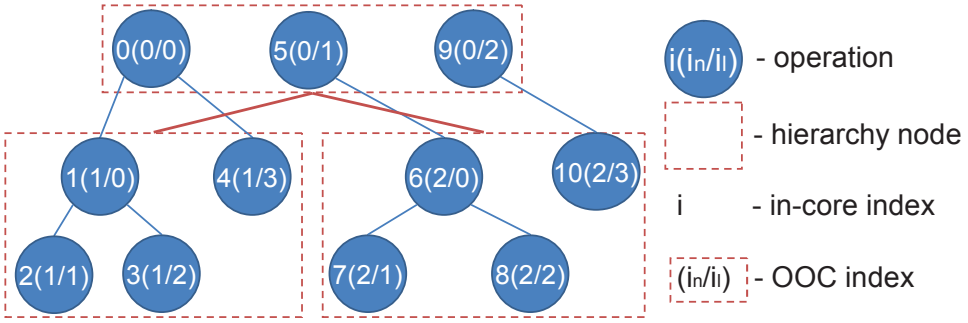

Figure 5.23: Split/collapse operation hierarchy represented as a forest of binary trees. The operations are shown in blue and the bounding volumes in red.

Connectivity Coding

In theory the proposed algorithm do not need to store the FVID of v_u as the difference to v is simply one plus the number of split operations in the subtree of v_t. If no split for v_t exists, the algorithm thus know that the offset is one and do not need to store it. In practice, not storing the FVID of v_u in other cases would require to traverse the whole subtree. As it is then at least two, the algorithm store the offset minus two to prevent the traversal. The triangle indices ($\text{FVID}_{0..2}$) are sorted such that the first two are children of v_t and v_u respectively. Then these two are encoded as differences to the FVID's of v_t and v_u. While the proposed algorithm could also encode the topology modifications of v_t and v_u bit-wise, as proposed by Kim and Lee [KL01]. This would however not be optimal as the tree is not balanced. The third vertex cannot be a child vertex of v. Its FVID is either less than v or at least the next higher FVID v_n of the currently active vertices. In the first case, a negative offset to v is stored and in the second case, a positive offset to v_n. The offset vo_3 is then again mapped to a positive value as described above. If more than one triangle is generated by the operation the algorithm can exploit an additional degree of freedom. First the triangle with the smallest third vertex offset vo_3 is compressed. The

successive triangles are sorted by increasing vo_3 and only the difference to the previous offset is stored. In addition to the indices, the proposed algorithm also store the number of faces to support non-manifold meshes and improve the compression rate for boundary edges.

Refinement Criteria

As no high accuracy is required for back-face culling, $\sin\alpha$ is quantized to eight bits. Due to the shrinking neighborhood, $\sin\alpha$ becomes smaller for the successive splits. The proposed algorithm thus encode the difference to the parent operation to exploit this fact. A separate probability table is used as the distribution is nevertheless not centered at zero but at a model dependent value.

The proposed algorithm encode ϵ_g and ϵ_a together as the screen space error ϵ_s is a combination of these two:

$$\epsilon_s = max\left(\epsilon_a, \epsilon_g(\mathbf{d}\cdot\mathbf{n})\right), \tag{5.20}$$

where $\mathbf{d}$ is the view direction and $\mathbf{n}$ the normal. The geometric error is only relevant if it is greater than the attribute error and the algorithm thus encode the maximum error ϵ and the ratio μ of ϵ_a to ϵ_g. Then the algorithm can then simply clamp μ to be at most one. Similar to the normal cone angle, the probabilities significantly deviate from a normal distribution. Thus the algorithm also encode μ quantized in a single byte with its own probability table. The maximum error can vary over a huge range. The upper bound depends on the model size and its lower bound is the quantization step q. To compactly encode this range, the proposed algorithm exploit the fact that only a rather low accuracy is required. In the current implementation the algorithm use a relative accuracy of 2% which is expressed by the following coding function:

$$\epsilon_{enc} \;=\; \left\lceil \frac{\ln\epsilon - \ln q}{\ln 1.02} \right\rceil, \tag{5.21}$$

where ϵ_{enc} is the encoded simplification error. Similar to $\sin\alpha$ the simplification error is monotonously decreasing and the algorithm also encode the difference to the parent operation. In contrast to $\sin\alpha$ and μ, the difference exhibits a normal distribution.

Note that the refinement criteria are stored as the first four bytes of the operation since they need to be decoded first to evaluate the necessary operations. The ID of v_u and the number of faces are stored directly after the refinement criteria as they are also required when checking for and preparing the required operations.

Attributes

To prepare the attributes for compression, the proposed algorithm first quantize each coordinate. The quantization step is chosen depending on the progressive mesh to be

encoded. The algorithm calculate the root mean square attribute difference $\overline{\sigma}_i$ for each attribute i over all operations and then use $q = a_{scale}\overline{\sigma}_i$ as quantization step. In my implementation the algorithm chose $a_{scale} = \frac{1}{16}$ which is a reasonable trade off between accuracy and compression rate. Assuming the vertices were collapsed to their midpoint, the algorithm use second order prediction for Δv_u (i.e. $-\Delta v_t$) and store the difference. More sophisticated estimations using subdivision schemes are not possible as this would again require at least the adjacent vertices to be present.

Note, that v_t and v_u of each split operation can be swapped without altering the encoded progressive mesh. The proposed algorithm exploit this to improve the compression rate. All operations are checked and if the total size of the encoded model is reduced when swapping the vertices, the swap is performed. The size reduction can be due to the different vertex attributes (especially for half-edge collapses), the FVID of the generated vertex, and – especially for operations close to the base vertices – due to smaller FVID offsets. To exactly calculate the compression improvement the algorithm would need to compress the complete progressive mesh twice for each operation, leading to a total complexity of $O(n^2)$. Swapping an operation however only influences the geometry offsets of the operation itself and the FVID offsets of all operations on neighbor vertices. Using an fixed approximate probability table, the algorithm can estimate the first one in $O(n)$ time for all operations. The neighborhood for the FVID offsets grows by a factor of $\sqrt{2}$ with each level of the hierarchy. Thus the total number of influenced operations throughout the whole hierarchy is: $\sum_{i=0}^{\log_2 n} 2^i \sqrt{\frac{n}{2^i}} = O(n)$. Starting from leaf level, the algorithm collect all operations that generate neighboring triangles and propagate them up to the root operations. During this process, triangles that lie completely between the two collapsed vertices are stored at that operation. After searching the influenced operations, the algorithm can estimate the effect of the swap on the compressed size of the FVIDs in total linear time as well.

5.4.3 Runtime Algorithm

Besides classifying the vertices before applying the operations, the adaption is split into first applying all collapse operations. Then the memory for the new vertices and faces is allocated while the data of collapsed ones is freed. Finally, the split operations can be performed by updating the vertices and then the current vertex indices. The dynamic data structures required for rendering are the vertex buffer containing the position and attributes and the index buffer storing the connectivity of the adapted mesh. Both are stored as vertex buffer objects (VBOs) and are separated from all other data. Table 5.12 gives an overview of all dynamic data structures.

5.4.3.1 Vertex State Update

In the first step the necessary operations are determined. If the vertex v needs to be split according to its refinement criteria, its number of child vertices in the next iteration need

buffers	**elements**	**memory (bytes)**
active faces	index VBO	$24m$
	FVIDs	$24m$
active vertices	vertex VBO ($\times 2$)	$8km$
	vertex ID ($\times 2$)	$8m$
	split & collapse cache ($\times 2$)	$16m$
	collapse target ($\times 2$)	$8m$
	next split & collapse ($\times 2$)	$16m$ [a] / $24m$ [b]
temporary	vertex count	$4m$
	face count	$4m$
	vertex prefix sum	$4m$
	face prefix sum	$4m$
total	in-core / out-of-core	$(112 + 8k)m$ / $(120 + 8k)m$

Table 5.12: Elements of the dynamic data structure. k and m are the number attributes and adapted mesh vertices. Next split and collapse are stored with 32 bits in the in-core (a) and 48 bits in the out-of-core case (b).

to be set to two. Additionally the number of faces that are created by this operation is decoded. Otherwise the number of vertices is set to zero if the refinement criteria allow a collapse, or to one if not. The collapse of a vertex is only possible if its corresponding and its target vertex v_t have not performed further splits. This can efficiently be checked by keeping the vertices sorted based on their FVID. In this case the previous vertex of v needs to be its target v_t and the target of the next vertex must not be v.

Two or three refinement criteria are checked for each active vertex for in-core and out-of-core rendering respectively. The most simple one is view frustum culling as a vertex can be collapsed if it lies outside the view frustum regardless of the simplification error. To prevent foldovers, all vertices that are outside of the view frustum are however not simply collapse, but modify the distance d of these vertices for the following LOD selection:

$$\tilde{d} = \left(c_{LOD} \left(\frac{\max(|x|,|y|,|z|)}{w} - 1 \right) + 1 \right) d, \tag{5.22}$$

where x, y, z and w are the homogeneous coordinates of the vertex after projective transformation and c_{LOD} a constant value. In the experiments $c_{LOD} = 100$ is used for a smooth LOD falloff outside the view frustum which prevents foldovers and popping artifacts when rotating or panning. The next test is back-face culling. The vertex is culled if

$$\mathbf{n} \cdot \mathbf{d} \quad > \quad \sin \alpha, \tag{5.23}$$

where $\mathbf{d}$ is the normalized view direction, $\mathbf{p}$ the vertex position, $\mathbf{n}$ the normal, and α the normal cone angle as discussed in Section 5.4.2.2. In addition occlusion culling is used for

out-of-core rendering based on the visibility of the bounding volumes. The vertex can be collapsed if its split or collapse is stored in an occluded bounding volume. The hardware occlusion queries [CCG*01] are used to determine which bounding volumes are visible. The queries are performed after rendering the complete scene and the results are fetched before the next adaption. This way the visibility is lagging behind by one frame but this is unproblematic as it is not used for rendering but for LOD selection only. Regardless of the query results the complete adapted mesh is always rendered. The simplification error is evaluated after culling. For this the first four bytes of the operation are required. Each active vertex has an eight byte cache storing the split and collapse refinement criteria to reduce decoding time and unaligned global memory access. The split cache only needs to be updated if the vertex was modified or created by a split operation in the previous frame. The complete vertex update is shown in Algorithm 12.

```
foreach vertex v in parallel do
    update_split_cache(v)
    set_vertex_cnt(v, 1)
    set_face_cnt(v, 0)
    if need_split(v)
        decode_number_of_faces(v)
        set_vertex_cnt(v, 2)
    elif need_collapse(v)
        target = get_target(v)
        target_next = get_target(next(v))
        if prev(v) == target && v != target_next
            set_vertex_cnt(v, 0)
```

Algorithm 12: Parallel vertex state update algorithm.

5.4.3.2 Parallel Edge Collapses

To perform a collapse, the algorithm need to check whether the target v_t of the current vertex v_u was not marked for splitting in the first stage. The collapse operation then simply moves vertex v_t to its old position v and copies the collapse cache of v_u to the split cache of v. Removal of vertex v_u and the degenerate faces are handled in later stages. Algorithm 13 shows the parallel processing of the edge collapse operations to prepare removal of the collapsed vertices and faces.

```
foreach vertex v_u in parallel do
    v_t = get_target(v_u)
    if !marked(v_t, split)
        collapse_vertices(v_t, v_u)
    else
        set_vertex_cnt(v_u, 1)
```

Algorithm 13: Parallel edge collapse algorithm.

5.4.3.3 Memory Management

Memory must be reserved for the additional vertices and faces before the split operations can be applied. While the ordering of faces is irrelevant for the algorithm, the vertices must be sorted by their FVID as discussed above. If the index and FVID buffers are large enough, the faces can be directly appended. To determine the position of the faces in the index buffer, the parallel prefix sum [SHZO07] of the number of generated faces is computed. After calculating the prefix sum, the total number of faces after all split operations can be determined. If the size of the face buffers is too small or significantly too large, new buffers are allocated and the content of the old ones is copied into them. When a reallocation is performed, the buffer size is set to the number of faces n_f plus a user-defined threshold n_{alloc}. If the buffer is larger than $n_f + 2n_{alloc}$ it is reduced to $n_f + n_{alloc}$. While the buffer resizing is applied to the vertices as well, new vertices cannot be simply appended to the vertex buffer. They need to be sorted by their ID as discussed above. This means that they have to be inserted after the corresponding split vertex. The algorithm accomplish this by copying the old vertices into a new buffer. During this process, the collapsed vertices are also removed by calculating the parallel prefix sum of the vertex count to determine the positions in the new buffer. The vertex IDs, caches and next split/collapse buffers need to be processed this way as well. While the memory of the old buffers could be freed after this step, the repeating allocation would drastically reduce the performance. The reorganisation and compaction of the vertex buffer are shown in Algorithm 14.

```
face_sum = prefix_sum(face_cnt)
if need_face_buffer_resize()
    resize_face_buffers()
vertex_sum = prefix_sum(vertex_cnt)
if need_vertex_buffer_resize()
    resize_vertex_buffers()
foreach vertex v in parallel do
    new_pos = vertex_sum[v]
    next_pos = vertex_sum[v+1]
    if new_pos != next_pos
        copy_vertex(v, new_pos)
```

Algorithm 14: Memory management algorithm.

5.4.3.4 Parallel Vertex Splits

The split operations are performed after memory allocation and reorganisation of the vertex buffer. To improve thread utilization the algorithm first compact the splits [SHZO07] such that each thread performs an operation. Every operation generates a new vertex v_u and moves v to its new position v_t. Additionally, the new faces are added to the index and the FVID buffers. For this the current indices for each of the new faces need to be determined.

Fortunately, the first two vertices of each new face are known as they are v_t and v_u. The third vertex needs to be located in the vertex buffer. By construction it is the vertex with the greatest FVID less or equal to the one stored in the face. The Binary Search is used to find this vertex in the vertex buffer. Note that while this does not fully utilize memory bandwidth it keeps the threads within every warp running in parallel which is also important for performance. Algorithm 15 shows the parallel vertex split.

```
compact(splits)
foreach split vertex v  in parallel do
    v_u = v + 1
    split_vertex(v, v_u)
    append_faces(v)
```

Algorithm 15: Parallel vertex split algorithm.

5.4.3.5 Index Update

The indices of the faces adjacent to split and collapse vertices need to be updated (e.g. $f_{n1} - f_{n6}, f_l$, and f_r in Figure 5.18) after performing all operations. This is necessary as the vertices of adjacent faces can perform their operations in parallel. The correct vertex can either be the previous one, the current one, or the next vertex in the sorted array. The first case occurs when the vertex was collapsed, while the last one occurs when the vertex was split. The second case can either happen when no operation was performed or the operation did not change the connectivity of that face. Algorithm 16 shows the parallel index update.

```
foreach indices i  in parallel do
    ID = get_vertex_id(i)
    FVID = get_final_id(i)
    if ID < FVID
        set_vertex_id(i, ID + 1)
    elif ID > FVID
        set_vertex_id(i, ID - 1)
```

Algorithm 16: Parallel index update.

5.4.3.6 Buffer Compaction

The final step of the adaption is the compaction of the index buffers to delete degenerate faces. Note that as the index VBO is used for rendering it needs to be compact anyways. For the buffer compaction a specialized in-place compaction algorithm from the section 5.2 is used, since the ordering does not need to be preserved. The main advantage besides a minor speedup is that these buffers need not to be duplicated.

5.4.3.7 Out-of-Core Memory Management

An additional memory management of the static data structures is performed for out-of-core rendering. Only the currently necessary bounding volumes are kept in graphics memory based on a priority scheduling. All relevant data is stored in graphics memory and the main memory consumption is minimal, as the complete algorithm runs on the GPU. In addition to temporary memory for loading, only the mapping of bounding volumes to memory positions, the file offsets required for out-of-core management, and the bounding boxes are stored in main memory. For loaded bounding volumes the pointers to their data in device memory are also stored. A bounding volume is required when at least one of the active vertices has a reference to it. A split operation can create vertices that reference volumes not available in graphics memory and the data needs to be loaded from disk. Each BV contains operations with different simplification errors. Based on the maximum error of the operations stored in a volume, a distance d_n beyond which no split is ever necessary can be derived. Then a priority $p = \frac{d_n}{d_v}$ is calculated, where d_v is the distance between the viewer and the bounding box. The data of the bounding volume is only required if its priority is at least one. The bounding volumes with higher priority are loaded first if the transfer from disk to graphics memory is not fast enough. This has the advantage that the model is uniformly adapted and no LOD starvation can occur. To limit the memory consumption, a maximum number of nodes n_{max} kept in graphics memory is specified by the user. When rendering several progressive meshes, the node memory is shared among all models. When the user moves through the scene, the visibility and the required LOD of the object change. This results in a continuous change of the bounding volumes currently required in graphics memory. As discussed above, loading data results in a visible delay of the adaption. This problem is solved by not only loading the currently required nodes, but also the nodes with lower priority, as long as enough space is available. Before uploading the currently required bounding volumes to graphics memory the unnecessary ones are first removed, until enough space is available. Since accessing the hard disk causes high delays, loading operations into main memory is preformed in a second thread. As soon as the data is available in main memory, the rendering thread can copy it into graphics memory after scheduling the occlusion queries.

5.4.4 Results

The test system consists of a 3.333 GHz Intel Core i7-980X CPU with 6 GB DDR3-1333 main memory, 16 lanes PCIe 2.0 slot, and an NVIDIA GTX580 (841/4204MHz) graphics card. OpenGL is used for rendering and CUDA for the adaption algorithm. The out-of-core data is stored on a SATAII hard disk (8.5ms/64MB/7200rpm) with approximately 100 MB/s read speed. The bounding volume data size is set to 64 kB, as host to device copy of blocks with up to this size is asynchronous. The resolution of 1920×1080 with a screen space error of 0.5 pixel is used. At most $n_{max} = 4096$ (256 MB cache) for all out-of-core

models is used. Table 5.13 gives an overview of the progressive meshes, which are tested in the experiments. All models use position and normal as vertex attributes ($k = 6$). The original meshes contain v_{max} vertices and f_{max} faces. The number of base mesh faces is zero and that of base mesh vertices v_0 is very low. The number of operations is $v_{max} - v_0$. The resulting file sizes and compression rates for the in-core (ic) and out-of-core (ooc) case are listed in Table 5.13 together with the number of BV nodes and bytes per vertex (bpv). Compared to in-core, the out-of-core static data requires approximately 1 additional byte for each operation. On average 1.2 bpv are consumed by the refinement information and 1.0/2.0 bpv by the tree structure in the in-core and out-of-core case. The connectivity and geometry need 2.4 and 5.8 bpv and 0.6 bpv are wasted due to the per operation compression. Additionally, 35 bytes main memory (bounding box, visibility, offset, and file position) and 5 bytes graphics memory (visibility and offset) are required per BV node. The compression performance is approximately 7 kop/s (in-core) and 4.3 kop/s (out-of-core). In the first case, approximately half of the time is required for the optimization and the other half for the two arithmetic coding runs. In the second, three additional arithmethic coding runs were required until convergence for all models.

Table 5.14 also lists the number of rendered faces, the total rendering time, and the memory consumption for the views shown in Figure 5.24, and the scene in the accompanying video, where the numbers are taken from the most complex frame. The ratio compared to an indexed face set (IFS) of the original model is shown in parenthesis. During rendering, the dynamic data structures consume additional memory. The total amount of graphics memory nevertheless stays below that of the original models. The out-of-core algorithm facilitates occlusion culling and out-of-core memory management, therefore the frame time is approximately 10% higher compared to in-core rendering although the number of faces is approximately 10% lower. The culling overhead is low because the number of bounding volumes is small compared to the number of triangles, only asynchronous queries are used, and only require a single switch between rendering and occlusion queries per frame.

model	v_{max}	f_{max}	*orig. (MB)*		*PM (MB)*	BV nodes	bpv	compr.	time
Dragon	3,609,455	7,218,906	165.2	ic:	43.0 (26.1%)		12.5		9m
				ooc:	48.2 (29.1%)	843	14.0		14m
Statuette	4,999,996	10,000,000	228.8	ic:	58.7 (25.7%)		12.3		12m
				ooc:	64.9 (28.3%)	1136	13.6		19m
Lucy	14,027,872	28,055,742	642.2	ic:	152.9 (23.8%)		11.4		37m
				ooc:	168.6 (26.3%)	2965	12.6		56m
David	28,184,526	56,230,343	1288.6	ic:	303.3 (23.5%)		11.3		1h 10m
				ooc:	335.2 (26.3%)	5945	12.5		1h 49m
St.Matthew	186,810,938	372,422,615	8537.8	ooc:	2038.0 (23.9%)	36034	11.4		11h 54m
Atlas	254,837,027	509,674,062	11665.5	ooc:	3039.7 (26.0%)	53159	12.5		16h 27m
Scene	492,469,814	983,601,668	22528.1	ooc:	5694.6 (25.3%)	100082	12.1		

Table 5.13: Progressive meshes examined in the experiments and compression results.

model		*rendered # faces*	*memory (MB)*	*frame time (ms)*	*MTPS*
Dragon	ic:	1,301,757 (18.0%)	152.3 (92.2%)	6.5	200.3
	ooc:	1,023,843 (14.2%)	142.2 (86.1%)	5.9	173.5
Statuette	ic:	1,479,282 (14.8%)	180.9 (79.1%)	7.2	205.4
	ooc:	1,338,246 (13.4%)	182.4 (79.7%)	7.3	183.3
Lucy	ic:	2,804,876 (10.0%)	379.9 (59.1%)	11.6	241.8
	ooc:	2,672,566 (9.5%)	397.3 (61.9%)	13.5	198.0
David	ic:	2,954,923 (5.2%)	541.8 (42.0%)	12.1	244.2
	ooc:	2,859,350 (5.1%)	498.0 (38.6%)	14.3	200.0
St.Matthew	ooc:	3,412,032 (0.9%)	541.7 (6.3%)	15.3	223.0
Atlas	ooc:	4,025,064 (0.8%)	605,7 (5.2%)	17.8	226.1
Scene	ooc:	5,514,913 (0.6%)	795.8 (3.5%)	39.7	138.9

Table 5.14: Rendering statistics of the experiments.

The algorithm can process up to 244/226 million triangles per second (MTPS) for static views in the in-core and out-of-core case respectively. The frame time linearly increases with the number of faces. This face count converges to a constant value with increasing model size which is a typical behavior of all LOD algorithms. Therefore, the frame time converges to a constant value as well. The same holds for the memory consumption of the out-of-core algorithm. As the test GPU can render an indexed face set with 600 MTPS, the performance of the proposed method is faster than rendering the original model as soon as 60% of the faces are removed. Figure 5.24 also shows the coarsening of culled faces.

The adaption and rendering time together with the memory consumption and number of faces for a pre-recorded movement through the scene are shown in Figure 5.25. The consumed memory is always below 796 MB, the frame rate is constantly above 30 frames per second (fps) with an average of 50 fps and 140 MTPS. The proposed approach quickly reacts to changes of the view with fast adaption of the scene complexity. Due to the high adaption performance no popping artifacts are visible in the video despite fast movements and the screen space error of 0.5 pixel is always achieved.

5.4.4.1 Discussion

In Table 5.15 the proposed algorithm are compared with three different types of approaches. The algorithm can render 180/140 MTPS (in-core/out-of-core) for dynamic views and up to 23/13 MTPS can be generated. Note that the number of generated triangles of the out-of-core algorithm is limited by the HDD speed. Due to the slightly reduced number of triangles for identical views, the relative rendering performance is 155 MTPS with up to 14.44 MTPS generated. The progressive mesh requires 11-14 bytes per vertex (bpv).

While compression approaches of course achieve better compression ratios, they have significant shortcomings regarding rendering. First of all, most of them do not support

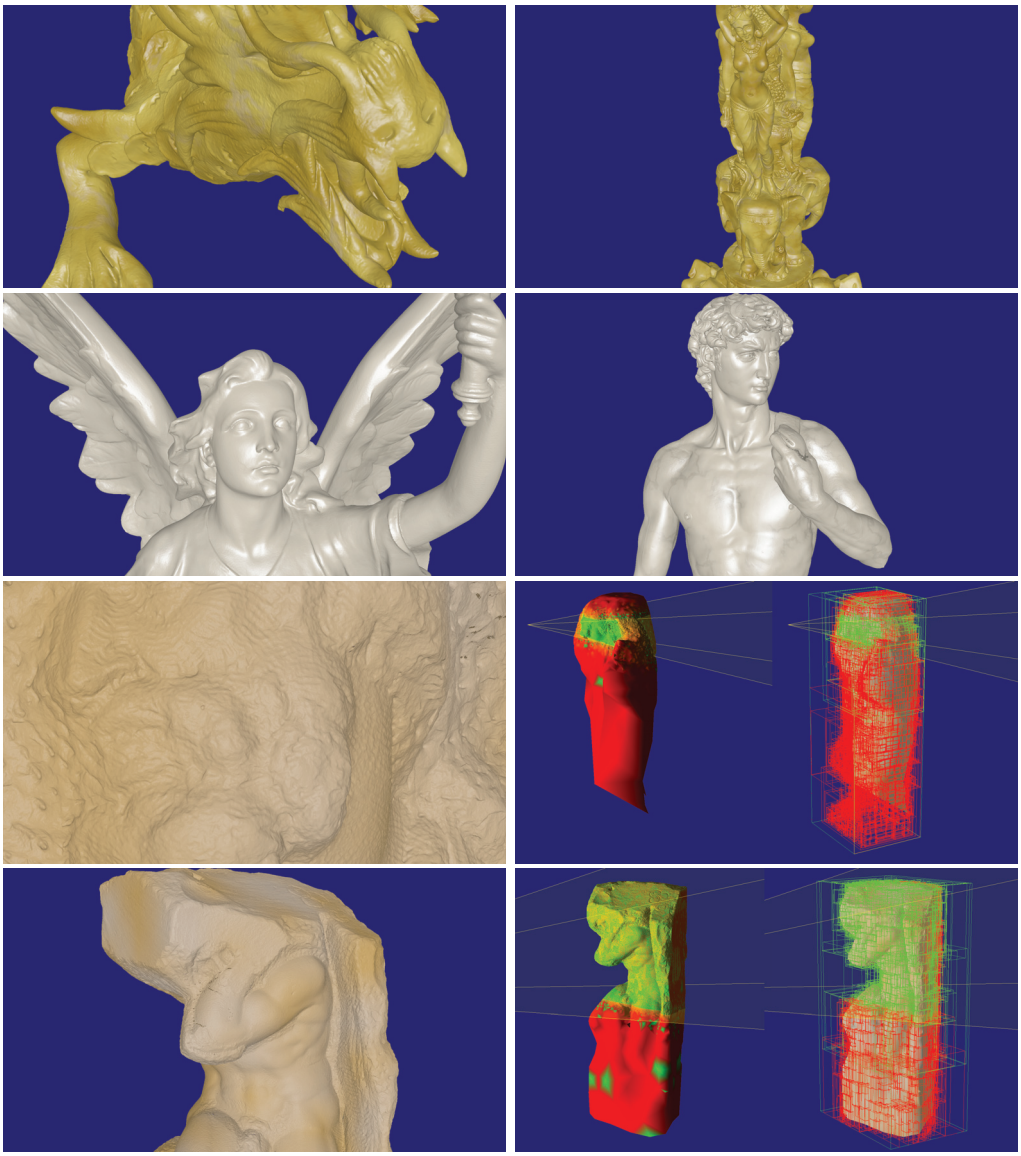

Figure 5.24: Renderings of view-dependently refined meshes. The external views show the view frustum (yellow), LOD (red: low; green: high), and the nodes used for occlusion culling (red: occluded; green: visible).

extracting a level-of-detail and thus only support view frustum culling (VFC) [CKLL09, CH09]. Others only allow simple level-of-detail schemes based on regular vertex clustering [DJCM09, JGA09]. This is however known to require at least an order of magnitude more primitives to achieve the same quality. Note that high compression ratios are also achieved by not encoding vertex normals which also reduces the visual quality as the normals computed from a simplified mesh can drastically differ from correctly simplified ones. Another problem is the complex connectivity coding that prevents parallel decompression. The fastest compression approach only achieves decoding of up to 0.3 MTPS.

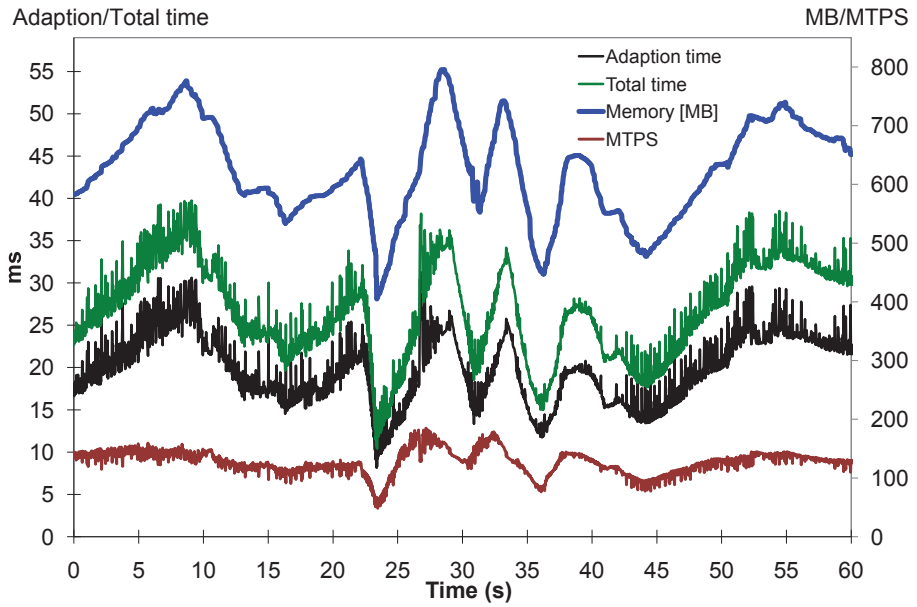

Figure 5.25: Timings, memory consumption and triangle rate for the scene using a pre-recorded camera path.

Considering the increased number of triangles compared to the proposed approach, the relative adaption performance is less that 30,000 triangles per second so it at least 500 times slower. Once generated, the resulting mesh can be rendered at full performance. Again it need to consider the tenfold increase in model complexity which translates to a relative performance of less than 60 MTPS or less than 40% compared to the proposed algorithm. So the proposed approach can render the same view more than twice as fast and can adapt the LOD more than two orders of magnitude faster.

Hierarchical level-of-detail (HLOD) algorithms also generally achieve high rendering performance unless special shaders are used [GBK03, GM05]. Compared to view-dependent progressive meshes, the number of primitives is however drastically increased. This is due to three reasons: First, the LOD is only evaluated per node of the hierarchy. This already doubles the number of primitives considering that there is usually a resolution factor of two between successive level. The second reason is that the LOD is only distance instead of fully view-dependent which prevents coarsening of back-facing and non-silhouette triangles. This also approximately doubles the number of primitives. Finally, special care must be taken at the node boundaries which also slightly increases the primitive count. In total, the number of primitives is 5 to 7 times higher for the same view [GBK03, CGG*04]. This factor can exceed 10 if vertex clustering is used [GM05]. The number of generated triangles either depends on the hard disk speed or the mesh decompression. On the used system, between 3.4 and 4.6 MTPS can be generated. Due to the higher primitive count, these are reduced to a relative performance of at most 1.47 MTPS [GBK03] (10% of the adaption performance of the proposed algorithm). The relative rendering performance

Algorithm	relative #triangles	rendered MTPS	generated MTPS	bpv
Proposed ic	1	180	23	11-13
Proposed ooc	0.9	140 (155)	13 (14.44)	11-14
Compression approaches				
[CKLL09]* ooc	VFC only	600 (n.a.)	0.07 (n.a.)	1-3
[CH09]* ic	VFC only	600 (n.a.)	0.4 (n.a.)	2-4
[DJCM09]* ooc	>10	600 (<60)	0.04 (<0.004)	1-3
[JGA09]* ooc	>10	600 (<60)	0.3 (<0.03)	2-4
HLOD approaches				
[GBK03] ooc	∼5.5	400 (73)	4.6 (0.84)	26
[CGG*04] ooc	∼6.5	500 (76)	4.6 (0.71)	33
[GM05] ooc	>10	190 (<19)	3.4 (<0.34)	61
Progressive Meshes				
[YSGM04] ooc	∼2.5	90 (36)	0.015 (0.006)	79
[HSH09] ic	∼1.05	30 (28)	0.8 (0.76)	69
Section 5.2 ic	∼1.2	80 (66)	4.4 (3.66)	22
Section 5.3 ooc	∼1.1	60 (55)	4.0 (3.63)	24

Table 5.15: Comparison of triangle rate and memory consumption with previous approaches. The relative performance is shown in parenthesis. Results marked with * are results of the original authors scaled to the performance of the used system, while all other were measured.

is also reduced to at most 76 MTPS [CGG*04] or 49% of the proposed approach. In summary, the proposed approach renders approximately twice as fast and can react ten times quicker to view changes.

View-dependent adaption algorithms can better compete with the proposed approach regarding the number of primitives. The actual factor depends on whether view- or distance-dependent adaption is used. It also depends on the degree of neighborhood dependencies and lies between 1.05 [HSH09] and 1.2 section 5.2 with view-dependent adaption and 2.5 [YSGM04] otherwise. The rendering performance is significantly lower than for HLODs due to the continuous geometry changes and interleaving adaption with rendering. The relative rendering performance lies between 28 MTPS [HSH09] and 66 MTPS section 5.3 which is 16-37% of the proposed approach. The adaption performance of the GPU algorithms (up to 4.4 MTPS) is significantly higher than CPU algorithms [YSGM04] with only 15,000 triangles per second. The relative adaption rate is at most 3.66 MTPS which is 25% of the proposed method. Compared to these algorithms, the proposed method can render the same views three times faster and adapt the LOD four times faster.

5.4.4.2 Analysis

Finally, the runtime of each step of the adaption and rendering algorithm is analyzed in Figure 5.26. As the rendering performance is identical to rendering a static model with the

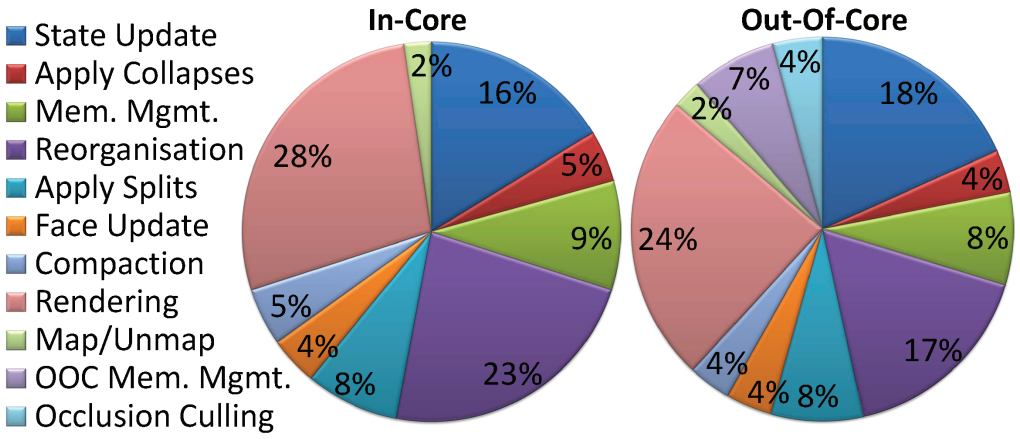

Figure 5.26: Relative time of the adaption steps compared to rendering.

same number of triangles, the proposed method needs approximately four times as long as rendering a static mesh. Considering that the proposed algorithm already cut down the vertices significantly due to the simplification of culled faces, the proposed method will almost always be faster than rendering the original model. The most expensive steps of the proposed algorithm are the state update and reorganisation. The state update is expensive because each active vertex needs to perform this step and the reorganisation as the algorithm need to maintain a sorted vertex buffer. The split/collapse cache reduced the state update time by 40%. Map and unmap are required for the mapping and unmapping of the index and vertex buffer for access from CUDA and can hardly be reduced or prevented.

5.4.5 Conclusion and Limitations

In this section an in-core and out-of-core dependence free progressive mesh representation was proposed. It is specifically designed for parallel view-dependent adaption and based on an implicit coding of topology modification inside the faces. In contrast to previous approaches no splits need to be postponed as they are waiting for others to be applied before them, which is otherwise very problematic for fast movements. Compared to progressive meshes and HLOD approaches the proposed algorithm reduce the rendering time, popping artefacts and the memory consumption significantly. This allows rendering of large models with fast movement nearly without popping artefacts. Compared to compression approaches the proposed algorithm require more disk space, but can keep compressed data in graphics memory. Further drawbacks of compression approaches are coarse grained random access and slow decompression, resulting in severe popping artefacts. Moreover, the refinement criteria and normals are not encoded. This improves the compression rate, but reduces the quality and significantly increases the number of faces.

The main limitation of the proposed algorithm is that the reorganisation of the vertices is rather expensive. An acceleration or prevention of this step would significantly increase the performance. Of course, the efficiency of the method soley depends on the underlying mesh simplification. If a model cannot be reduced using geometric simplification, other approaches (e.g. Fax Voxels [GM05]) are better suited.

CHAPTER 6

Progressive Mesh Editing

Highly detailed geometric models are very popular in interactive applications such as computer games. These models are usually represented as triangle meshes. To render several of these models at real-time frame rates, level-of-detail (LOD) techniques are commonly used. Mesh editing is used to correct errors or animate the models. In the traditional work flow simplification algorithms are used afterwards to generate the LODs. This requires a re-simplification after editing and restricts animations to space deformations. To solve this problem, multi resolution modeling approaches were proposed that create a set of LODs with increasing resolution. These subdivision meshes are comparable to static LODs that are simply a set of polygon meshes. Although multi resolution modeling is restricted to geometric deformations and cannot directly modify the connectivity, it is still a very valuable tool of the modeling pipeline.

In contrast to the static LODs used for previous multi resolution modeling algorithms, dynamic LODs store a coarse base mesh and a sequence of refinement operations. The most common data structure used in this context are progressive meshes. Using the proposed approach it is possible to edit the mesh at a lower quality while modifications are automatically propagated to finer resolutions. Thus only a few vertices need to be modified, to achieve large scale or small scale deformations, depending on the current LOD. In contrast to progressive meshes, multi resolution modeling requires a local geometry encoding to allow editing and the propagation of the geometric modifications. The proposed approach use a parallel simplification algorithm to generate the progressive mesh and to propagate changes to coarser resolutions. The main contributions of the proposed approach are:

- Real-time editing of large progressive meshes.
- Mesh simplification and progressive mesh generation on the GPU.
- A data structure that allows animated progressive meshes.

6.1 Related Work

Mesh editing has been an active field of research over the last three decades. Early approaches focused on editing of smooth surfaces while later ones tried to preserve local properties of the mesh. The algorithm of Welch and Witkin [WW94] is based on arbitrary triangle meshes and allows free-form shape design. Taubin [Tau95] later optimized this approach and improved the efficiency. Editing of smooth surfaces is still an active field of research [LF03].

6.1.1 Multi Resolution Modeling

Meshes that contain geometric details require special techniques to preserve these during editing. Such techniques mostly use multi resolution representations of the mesh. The details are stored relative to the coarser model for one or more quality levels [FB88, ZSS97, KCVS98, KVS99, GSS99, Gar99]. The user can edit the mesh on a lower quality level and the changes are automatically propagated to the higher levels. This way only a few vertices need to be edited to achieve large changes. When representing details of a surface in terms of so-called Laplacian or differential coordinates [LSCo*04, SCOL*04], the fine-scale surface can be reconstructed by solving a linear Laplacian system containing the modified differential coordinates. For the manipulation of a mesh Sorkine et al. [SCOL*04] proposed an editor that includes operations like the interactive free-form deformation in a region of influence (ROI) or the possibility to integrate detail of one surface into another. These transformations are performed on a Laplacian representation of the given surface. Marinov et al. [MBK07] mapped a multi resolution deformation framework to the GPU that does not support a direct transformation of the surface. Since the displacement vectors are encoded independently for each component and details are not preserved uniformly over the surface, visible artifacts in highly deformed regions can occur.

6.1.2 Mesh Simplification

Mesh simplification is one of the fundamental techniques for real-time rendering of complex polygonal models. There is an extensive amount of methods that mainly focus on accurate bounds of the simplification error. A review of simplification algorithms is given in a section 3.1 and 5.1.

6.2 Parallel Progressive Mesh Editing[38]

Traditionally, multi resolution editing performs global deformations by decomposing the geometry of a highly-detailed model into a coarse base surface and reconstructing

38 Submitted, Vision Modeling and Visualization 2012 (VMV2012) [DGG12].

the applied modifications using a displacement in normal direction. In contrast to this the proposed algorithm use edge split operations of a progressive mesh to propagate the transformations to finer resolutions. Therefore, an appropriate data structure that contains relevant informations about the connectivity and vertex attributes is proposed.

6.2.1 Overview

The operations are stored in a tree structure that is generated by a parallel simplification of the original mesh. Based on the operation tree the model can continuously be adapted using parallel vertex split and edge collapse operations. The proposed algorithm can be divided in two main phases:

1. Simplification of the model and generation of the progressive mesh operation tree.
2. Multi resolution editing of the progressive mesh.

To preserve the connectivity during the propagation of the modifications the local ordering of operations needs to be preserved. This implies for example that a collapse is only allowed, if no further vertex in any adjacent triangle is collapsed or split. The current resolution can be edited using a handle and a euclidean distance based region of influence (ROI). The handle can be translated and rotated and the transformation is applied to the whole ROI (see Figure 6.1).

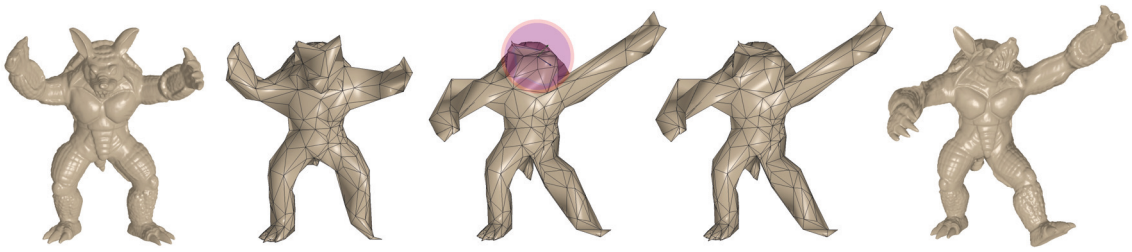

Figure 6.1: Editing of the Armadillo progressive mesh. Notice how the fine geometric details are preserved by the local encoding of the split operations.

6.2.1.1 Progressive Mesh

The original model is simplified by collapsing all non-conflict edges in parallel. The original model can then be reconstructed by performing a sequence of parallel vertex split operations. Figure 6.2 shows the principle of an edge collapse operation col_v and the corresponding split operation spl_v. By applying col_v to an edge defined by the vertices v_t and v_u it is contracted into the vertex v. The new vertex v is computed based on the neighboring triangles. The adjacent faces f_l and f_r of v_t and v_u degenerate and are removed. Since spl_v represents the inverse operation of col_v, the faces f_l and f_r are generated by splitting the vertex v into v_t and v_u. Additionally, an update of the connectivity between the vertices v_t and v_u and the adjacent faces is performed.

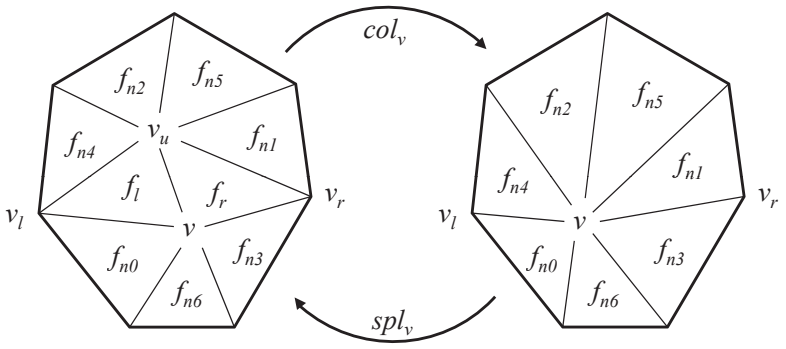

Figure 6.2: Edge collapse and vertex split operation (comp. Figure 5.1).

6.2.1.2 Operation Coding

The data structure used to encode the operations is based on the one proposed in the section 5.4. In this context, the main advantage is that non-manifold meshes are supported by storing the successive topology modifications within the triangles instead of the operations. For this the proposed algorithm need to generate so-called final vertex IDs (FVIDs) by enumerating the vertices after computing all edge collapses and storing them in the data structure. During a collapse operation an edge, which is defined by the vertices v_t and v_u, is contracted into v. Thereby, the vertex v_u disappears and v_t can be used as the collapse vertex v after updating the vertex position. In addition, for each operation the number of the vertices, which are collapsed into the current active vertex v is summed up and stored in the data structure. To compute the required FVIDs the algorithm start with the base mesh vertices and enumerate them with respect to their vertex count, that is stored in the corresponding operation. Then the FVIDs for the next finer level can be computed, where v_t receives the ID of the collapse vertex v and that of v_u is increased by the stored vertex count of v_t. This way all possible topology modifications are encoded within the vertex indices of the faces. The faces are then stored in the split operation where they are generated. The faces are encoded by storing their vertex $\text{FVID}_{0..2}$ for the finest resolution.

Figure 6.3 shows this numbering of the vertices. The leaf nodes of the binary tree forest are simply numbered from left to right. Then the collapsed vertex v receives the FVID of its left child v_t which is the smaller one. The resulting encoding of the faces is shown on the right side of the figure for a simple model. If a face is now decoded, when performing the split operation, the current vertices are either those with the same FVID, or the one with the greatest FVID smaller FVID_i. In the example, the active vertex for the FVID 1 when performing split 1 need to be found. The currently active vertex with the greatest FVID smaller than 1 is vertex 0 which is the collapse target of vertex 1.

In summary each operation encodes the attributes of v_t and v_u, the refinement criteria, the generated faces including later modifications and the subsequent operations. In contrast

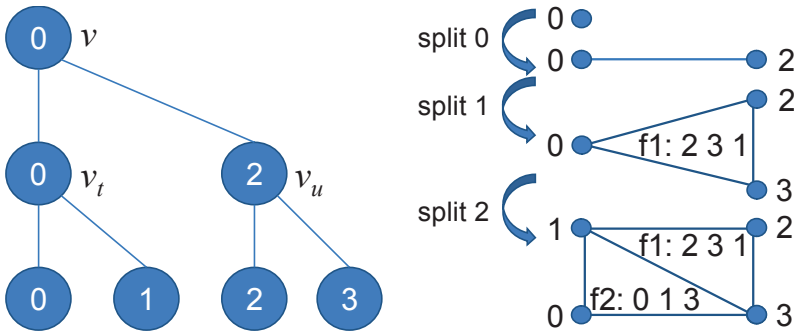

Figure 6.3: Computation of the final vertex IDs and encoding of the generated faces (comp. Figure 5.21).

to the algorithm proposed in the section 5.4 the operations are not compressed, but keep their uncompressed form to allow editing. The local ordering is preserved by storing the maximum of the simplification error and the errors stored for the neighbor vertices plus a small epsilon. This way only the errors of neighboring vertices need to checked to enforce a strict local ordering of split and collapse operations.

6.2.1.3 Local and Global Attributes

In compression approaches the vertex offsets are often encoded in the local coordinate system of the split vertex. While this improves compression rates, a transformation of the split vertex directly applies to all its decendents. To support a smooth propagation of the transformation to neighboring split vertices, the local coordinate system is averaged from all adjacent vertices (see Figure 6.4). Local position, normal and tangent (P_{int}, N_{int}, T_{int}) are computed as weighted average of v and all vertices adjacent to v_t or v_u. Then the position and normal offsets (P_{off}, N_{off}) are encoded in the local coordinate system spanned by N_{int} and T_{int}. Note that for the tangent only one degree of freedom remains and it is thus encoded as rotation about N.

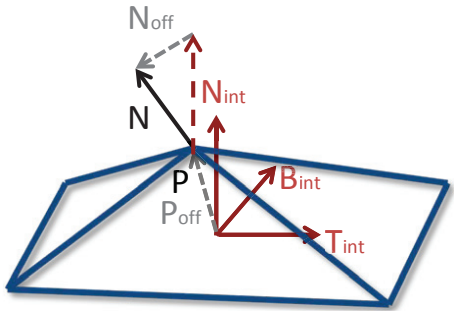

Figure 6.4: Vertex attributes encoded relative to the local coordinate system interpolated from neighboring vertices.

6.2.2 Progressive Mesh Generation

The progressive mesh generation is based on the parallel edge collapse simplification algorithm proposed in the section 3.2 which is restricted to generate a set of static LODs. Figure 6.5 gives an overview of the extensions and modifications necessary to construct a progressive mesh that allows both parallel adaption and real-time editing.

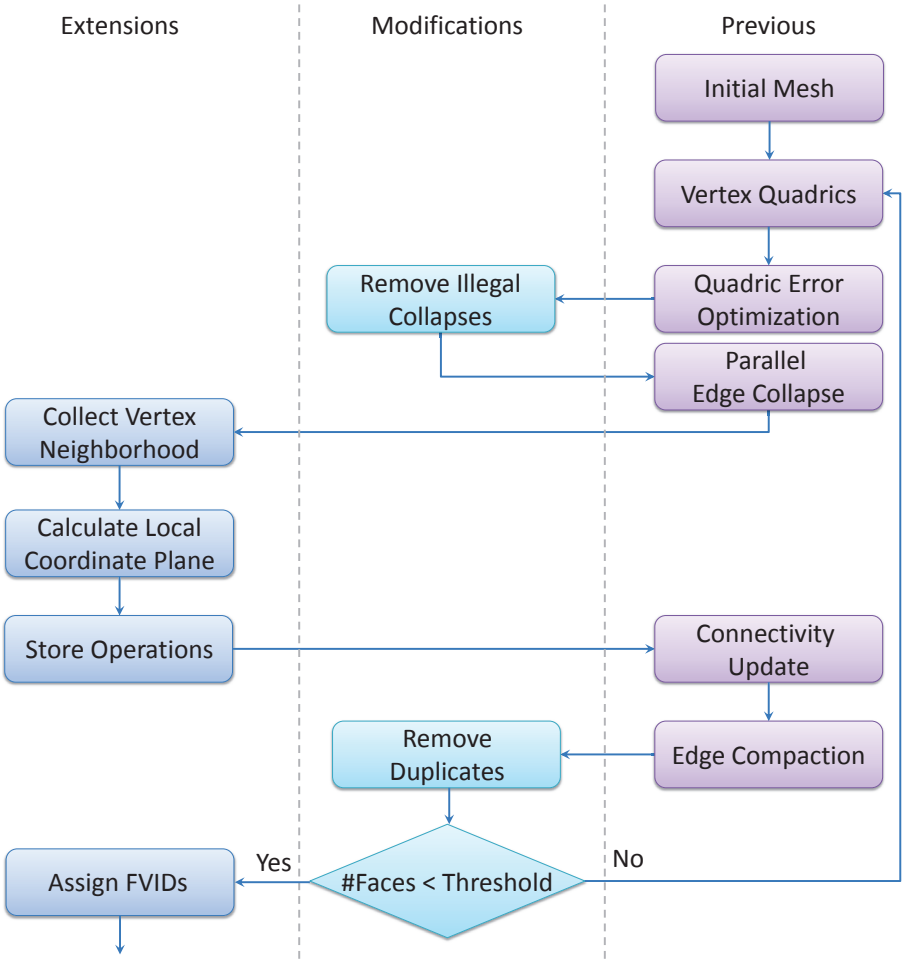

Figure 6.5: Progressive mesh generation including the extensions (left) and modifications (middle) of the previous simplification algorithm (right).

After loading the initial mesh from an indexed face set, the attributes and indices are transferred to the GPU and stored in a *vertex buffer* and *index buffer*. Then the edge data structure is filled as in the original simplification algorithm. Additionally, the edge indices for each face are stored since them are later required to guarantee a fixed collapse neighborhood. In contrast to the previous simplification algorithm the proposed algorithm use memoryless simplification [LT98] which results in computing the vertex quadrics inside the simplification loop.

All adjacent faces and boundary edge quadrics are accumulated to calculate the vertex

quadrics. The quadric error optimization is identical to the original method with the exception that the stored simplification error ϵ_v of vertex v is:

$$\epsilon_v = \max\left(\epsilon_{quadric}, (1+\epsilon_{float}) \max_{i \in neighbors(v)} \epsilon_i^s\right), \tag{6.1}$$

where $\epsilon_{quadric}$ is the quadric error, ϵ_i^s is the previous simplification error stored for vertex i, and the multiplication with $1+\epsilon_{float}$ assures that the error stored for collapsing v_t and v_u to v is larger than the error of previous neighboring collapses. This property is later used during adaption and modeling to guarantee a fixed neighborhood for each split and collapse operation. This also necessitates a modification of the overlapping collapse removal step that requires the additional edge references stored for each face. Fixing the neighborhood requires that two vertices sharing a common edge are not collapsed in parallel. For static LODs preventing a collapse of two adjacent edges was sufficient. In addition, the collapse can also not be performed, if v_l or v_r are not connected to v_u or v_t by another triangle in addition to f_l and f_r. This condition is necessary because otherwise v_l or v_r will be missing in the neighborhood of v_u and v_t when performing the vertex split.

After performing the collapses, the split operations can be stored in the progressive mesh data structure. This is done by first collecting the neighborhoods for all operations and then encoding the attribute offsets in the local coordinate system of the neighborhood. Finally, the connectivity is updated and the collapsed and duplicate edges are removed. The edge references stored in the faces are also updated during the compaction of the edge buffer.

When the number of faces falls below a specified threshold, the simplification loop is exited and the FVIDs are assigned to the vertices and split operations. The data structures required during simplification are listed in Table 6.1. Note that not all data are required during the complete algorithm and the corresponding buffers can be freed as soon as they are not used anymore.

6.2.3 Editing

Editing is based on modifying the attribute offsets of vertices encoded in split operations or the attributes of base mesh vertices. If a vertex is edited, its global attributes are changed and need to be mapped to modified local attributes. The editing is transferred to finer levels by encoding the split offsets in the local coordinate system of the neighbor vertices. For the coarse levels, the global attributes need to be recomputed after editing. Thus the proposed algorithm need to constantly convert between local and global attributes.

buffers	elements	bytes per entry
vertices	vertex VBO	$4k$
	vertex quadric	$2k^2 + 6k + 4$
	min edge ID	4
	min edge cost	4
	active flag	4
	adjacent vertex attributes	$4k$
	adjacent vertex count	4
	operation index	4
	split index	4
	FVID (x2)	8
edges	vertex index (x2)	8
	active flag	4
	collapse state	4
	optimal placement	$4k$
	temporary data	28
faces	index VBO	12
	active flag	4
	edge index (x3)	12
operations	vertex local attributes (x2)	$8(k + 1)$
	vertex index (x2)	8
	target vertex	4
	cost	4
	split index (x2)	8
	child count	4
	face index	4
	temporary	4
temporary	edge sort order (x2)	8
	edge sort key	4
	edge prefix sum (scan)	4

Table 6.1: Data structure used during mesh simplification, where k is the number of attributes.

6.2.3.1 Local and Global Attributes

The interpolated local coordinate systems of the neighbor vertices are used as reference coordinate system. As only the collapse target and its one ring are available only these can be used for interpolation. Therefore, it need to be guarantee, that the neighbor vertices are the same for a split and its inverse collapse operation. This is assured by the definition of the local ordering of the operations (see Section 6.2.1.2). Figure 6.6 shows the neighborhood used as reference for vertex v_t. Note that vertex v is weighted by a factor of two as it is closest to v_t.

Given the attribute offsets in the global coordinate system and the local coordinate

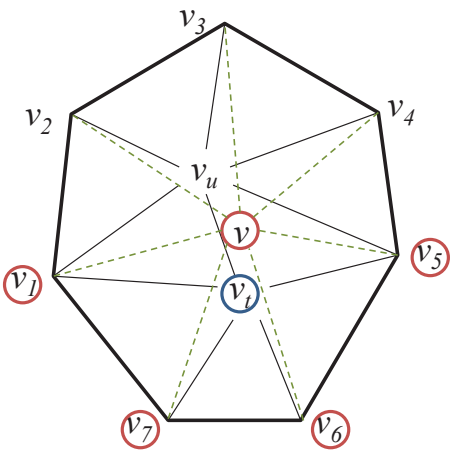

Figure 6.6: Vertices in the split neighborhood of v_t. The ones used to interpolate the reference coordinate system of v_t are marked in red.

system of the neighborhood, the local offsets (P_{local}, N_{local} and α) are calculated as:

$$P_{local} = (P_{off} \cdot T_{int}, P_{off} \cdot B_{int}, P_{off} \cdot N_{int})^t \tag{6.2}$$

$$N_{local} = (N_{off} \cdot T_{int}, N_{off} \cdot B_{int}, N_{off} \cdot N_{int})^t \tag{6.3}$$

$$T_{ortho} = normalize\,(T_{int} - N\,(N \cdot T_{int})) \tag{6.4}$$

$$\alpha = \arctan \frac{T_{ortho} \cdot (N \times T)}{T_{ortho} \cdot T}, \tag{6.5}$$

where P, N and T are position, normal and tangent of the vertex v_t or v_u, $B_{int} = N_{int} \times T_{int}$ the interpolated bitangent and α the rotation of T about the normal of the local coordinate system. The required position and normal offsets are:

$$P_{off} = P - P_{int} \tag{6.6}$$

$$N_{off} = N - N_{int}. \tag{6.7}$$

On the other hand, the global attributes of the vertex can be calculated as:

$$P_{off} = (T_{int} \quad B_{int} \quad N_{int})^t P_{local} \tag{6.8}$$

$$N_{off} = (T_{int} \quad B_{int} \quad N_{int})^t N_{local} \tag{6.9}$$

$$P_{global} = P + P_{off} \tag{6.10}$$

$$N_{global} = N + N_{off} \tag{6.11}$$

$$T_{global} = T_{int} \cos(\alpha) + (N_{local} \times T_{int}) \sin(\alpha). \tag{6.12}$$

6.2.3.2 Edit Propagation

After editing the changes need to be propagated to the coarser meshes. As noted earlier, the propagation to finer resolutions is handled automatically during the split operations. During editing, all modified vertices are marked by setting their modified flag to one. This value means that the local coordinate system and the global attributes where changed. Then the edited vertices are used for the calculation of the interpolated coordinate system. By adding the attribute offsets, which are encoded in the local coordinate system, finer details can be rebuild. For the propagation to coarser resolutions, the local coordinate systems of the modified vertices need to be recalculated during the collapses. As the proposed algorithm use memoryless simplification and guarantee a fixed neighborhood for every collapse, the edge quadrics can be simply recomputed. Then the quadric is again minimized to calculate the new vertex attributes in the coarser mesh. The new simplification error is also calculated as discussed in Section 6.2.2. Due to storing the attribute offsets in the local coordinate system of the neighborhood, the proposed algorithm need to recompute the collapse for every vertex that is adjacent to a modified one. This can be checked by simply traversing all triangles in parallel and setting the modified flag of each triangle vertex to two if it is zero and at least one of the others in the triangle has a modified flag of one. A modified value of two now means that the vertex has a new local coordinate system but still its old global attributes. Note that the proposed algorithm do not need to prevent race conditions here as it only change the flag from zero to two. The modification flag is then set for all vertices for which the collapse was recomputed to transfer the editing up to the base mesh. This implies that the progressive mesh has to be collapsed down to the base mesh before saving.

6.2.4 Adaption Algorithm

The adaption algorithm is based on the progressive mesh rendering algorithm proposed in the section 5.4. The algorithm is subdivided into several consecutive steps to implement the adaption on massively parallel hardware. The partitioning is required for thread synchronisation while each step can be processed completely in parallel. In the first step the proposed algorithm update the state of the vertices as in the original algorithm, but a global simplification error is used instead of view dependent refinement criteria during editing. Additionally, overlapping split and collapse operations are removed to guarantee the fixed neighborhood. This is necessary to assure that all for the operation required neighbors exist. Then all edge collapses are performed in parallel. Another modification is that the proposed algorithm recalculate the local coordinate system of the corresponding split operation if the vertex v was marked as modified (see Section 6.2.3.2). The memory management required before splitting is unmodified. For the split operations the proposed algorithm additionally need to calculate the global attributes of the vertex (see Section 6.2.3.1). Finally, the index update and buffer compaction of the original

algorithm are performed.

The dynamic data structures required for adaption and editing are listed in Table 6.2. The vertex buffer containing position and attributes and the index buffer storing the connectivity of the adapted mesh are required for rendering and thus separated from all other data. The modification flag is used to mark all vertices for which the split operations need to be updated. In the following the extensions of the progressive meshes adaption algorithm are discussed in detail.

buffers	**elements**	**bytes per entry**
active faces	index VBO	12
	FVIDs	12
active vertices	vertex VBO (×2)	$8k$
	vertex ID (×2)	8
	modified flag (×2)	2
	collapse target (×2)	8
	next split & collapse (×2)	16
temporary	vertex count	4
	face count	4
	vertex prefix sum	4
	face prefix sum	4
	vertex quadric	$2k^2 + 6k + 4$

Table 6.2: Elements of the dynamic data structure, where k is the number attributes and additions are marked bold.

6.2.4.1 Illegal Operation Removal

After updating the state of all active vertices, splits and collapses that cannot currently be performed are removed. The algorithm traverses all triangles in parallel and checks the vertex states again. For the split operations, only the vertex with the highest simplification error can be split in each face f. In addition, no vertex of f can be collapsed, if any other is marked for splitting. Finally, a collapse operation can only be performed, if its simplification error is the lowest in the triangle. The complete removal of illegal operations is shown in Algorithm 17.

```
foreach face f in parallel do
    if any_vertex_marked(f, split)
        simplification_error_max = get_max_split_error(f)
        unmark_dependent_splits(f, error_max)
    if any_vertex_marked(f, collapse)
        simplification_error_min = get_min_collapse_error(f)
        unmark_illegal_collapses(f, error_min)
```

Algorithm 17: Parallel removal of illegal operations.

6.2.4.2 Parallel Edge Collapses

When no neighboring vertices of the edge are marked, the collapse operation simply moves vertex v_t to its old position v. Otherwise, the weigthed average of the adjacent vertex attributes is required. This is calculated by counting the number of adjacent vertices and accumulating their attributes. If the vertex was marked as modified, the local coordinate system of the next split operation needs to be recalculated and the modified flag has to be propagated to the parent vertex and its neighborhood. The removal of vertex v_u and the degenerated faces are handled in later stages. Algorithm 18 shows the parallel processing of the edge collapse operations to update the operation after editing and to prepare the removal of the collapsed vertices and faces.

```
foreach face f in parallel do
    v1, v2, v3 = get_vertices(f)
    atomic_add(acc_adjacent_sum(v1), v2 + v3)
    atomic_add(acc_adjacent_sum(v2), v1 + v3)
    atomic_add(acc_adjacent_sum(v3), v1 + v2)
    atomic_add(adjacent_number(v1), 2)
    atomic_add(adjacent_number(v2), 2)
    atomic_add(adjacent_number(v3), 2)
    q = face_quadric(f)
    atomic_add(vertex_quadric(v1), q)
    atomic_add(vertex_quadric(v2), q)
    atomic_add(vertex_quadric(v3), q)
foreach vertex vu in parallel do
    v = get_target(vu)
    if marked(v, modified) || marked(vu, modified)
        LCS_VT = optimize_quadric(v)
        LCS_VU = optimize_quadric(vu)
        update_split(v, LCS_VT)
        update_split(vu, LCS_VU)
    else
        restore_attributes(v)
    collapse_vertices(v, vu)
```

Algorithm 18: Parallel edge collapse algorithm.

6.2.4.3 Parallel Vertex Splits

As in the algorithm proposed in the section 5.4 the split operations compact first to improve thread utilization. Then the global position of v_t and v_u need to be calculated from the weighted average of the local offsets stored in the operation. For this the weighted average of the adjacent vertex attributes is required again (see Section 6.2.1.3). For splitting, the new faces need to be added to the mesh at first. Then the global attributes are calculated. Algorithm 19 shows the parallel vertex split.

```
compact(splits)
foreach split vertex v in parallel do
    v_u = v + 1
    split_vertex(v, v_u)
    append_faces(v, face_sum[v])
foreach face f in parallel do
    v_1, v_2, v_3 = get_vertices(f)
    atomic_add(acc_adjacent_sum(v_1), v_2 + v_3)
    atomic_add(acc_adjacent_sum(v_2), v_1 + v_3)
    atomic_add(acc_adjacent_sum(v_3), v_1 + v_2)
    atomic_add(adjacent_number(v_1), 2)
    atomic_add(adjacent_number(v_2), 2)
    atomic_add(adjacent_number(v_3), 2)
foreach split vertex v in parallel do
    LCS_VT = acc_adjacent_sum(v) / adjacent_number(v_t)
    LCS_VU = acc_adjacent_sum(v_u) / adjacent_number(v_u)
    calc_attributes(v, v_u, LCS_VT, LCS_VU)
```

Algorithm 19: Parallel vertex split algorithm.

6.2.5 Results

The test system consists of a 3.333 GHz Intel Core i7-980X CPU with 6 GB DDR3-1333 main memory and an NVIDIA GTX580 (841/4204MHz). CUDA is used to implement the parallel simplification and editing and OpenGL for rendering. Table 6.3 and 6.4 gives an overview of the generated progressive meshes and the runtime performance during modeling. All models use position and normal as vertex attributes ($k = 6$). The original meshes contain v_{max} vertices and f_{max} faces. As the progressive meshes are uncompressed, they require approximately twice the memory than the original models (IFS). The maximum memory consumption is between 16 times (for small models) and 11 times (for larger ones) than that for the original model. This maximum is reached at the beginning of the simplification. During rendering the maximum is approximately $\frac{2}{3}$ since the edge data structures are not required anymore. Similarly to the simplification, the maximum is reached when the model is refined to full resolution. Compared to the approach proposed in the section 3.2 the preprocessing performance is lower by a factor of 5.8. The two main reasons for this are the larger neigborhood that reduces the number of parallel collapses by a factor of two and the generation of the progressive mesh data structure. In addition, the memoryless simplification is computationally more expensive than the normal quadric error simplification used in that algorithm. The adaption performance is even by a factor of 9.6 lower than that of the algorithm proposed in the section 5.4. This is partially again due to the fact that the proposed editing algorithm introduce neighborhood dependencies which reduce the number of parallel operations by a factor of approximately 6. On the other hand, the uncompressed data requires more memory bandwidth and the transformation from local to global coordintes also slows down the adaption. The

model	v_{max}	f_{max}	IFS
Horse	48,485	96,966	2.7 MB
Armadillo	172,974	345,944	7.9 MB
St. Dragon	437,645	871,414	19.9 MB
Welsh Dragon	1,105,352	2,210,673	50.5 MB
Dragon	3,609,455	7,218,906	165.2 MB

Table 6.3: Models used for evaluation.

	Simplification				Rendering		
model	PM	memory	time (s)	kOp/s	split kOp/s	collapse kOp/s	update kOp/s
Horse	5.4 MB	41.3 MB	0.3	162	595	443	384
Armadillo	19.1 MB	147.3 MB	0.6	288	1181	810	631
St. Dragon	48.3 MB	372.2 MB	1.4	313	1201	882	674
Welsh Dragon	122.3 MB	941.6 MB	3.2	346	986	381	336
Dragon	399.0 MB	1872.6 MB	10.4	347	760	379	335

Table 6.4: Simplification, adaption, and update performance.

reduction of the number of operations and thus decreasing performance for larger models is due to the fact that fewer operations were performed in parallel. The reason for this is that these models were not pre-simplified and therefore contained many coplanar faces. In the current implementation this blocks many collapses as only those are performed that have a smaller error than all of their neighbors. Although the proposed algorithm broke the possible deadlock by adding a small random number to the error, a more sophisticated solution would lead to more parallel collapses and thus increase the split and collapse performance.

The modeling performance of the proposed approach is approximately on par with algorithm of Marinov et al. [MBK07]. In contrast to their approach the proposed algorithm are however able to edit a mesh at different resolutions and the modifications are transferred directly to all LODs. Previous multi resolution modeling approaches like the method of Zorin et al. [ZSS97] are only suitable for models with subdivision connectivity. In Laplacian mesh editing [SCOL*04] a smoothed surface is used for editing and the topology is transferred back to the surface afterwards. The drawback of that approach is that modifications can only be performed for small ROIs with at most 100K vertices at interative frame rates. The proposed algorithm is on the other hand able to handle models containing up to serveral million triangles and edit them at real-time frame rates, independent of the ROI size. Figure 6.1 and 6.7 show some of the generated progressive meshes during editing.

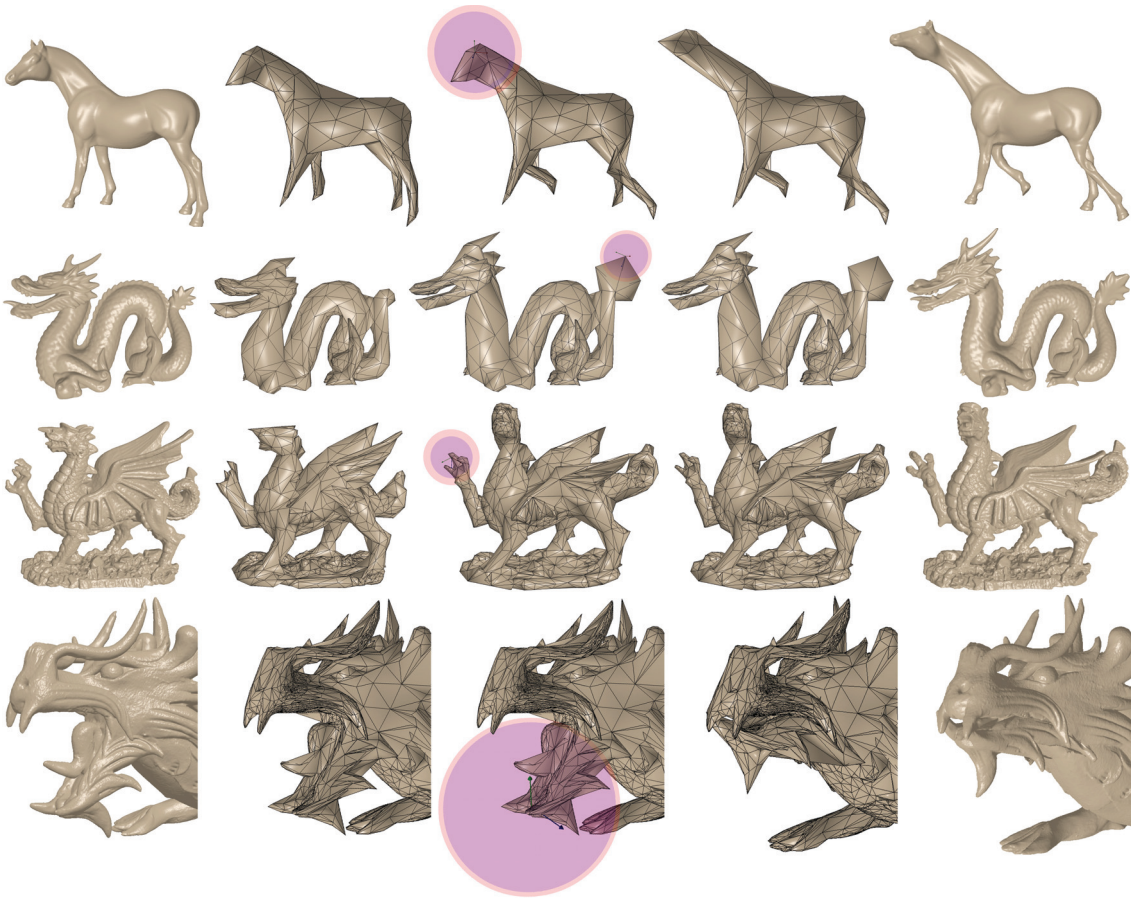

Figure 6.7: Several progressive meshes used in the evaluation. From left to right: original models, three frames captured during editing where the purple spheres show the ROIs, and the final proressive meshes refined to full resolution.

6.2.6 Conclusion and Limitations

In this chapter a parallel progressive mesh generation and editing algorithm was proposed. Its input is an indexed face set from which it first generates a progressive mesh data structure. The progressive mesh can then be edited at any resolution. The modifications are automatically propagated to finer resolution using an encoding of the split operations based on local coordinate systems. In addition to that the coarser resolutions up to the base mesh are updated using memoryless simplification. This leads to a valid progressive mesh during the complete editing pipeline.

The main limitation of the proposed algorithm is that the model size is currently limited to a few million triangles. In the future it can be expanded and improved for larger models by using out-of-core or possibly even compression techniques as in the algorithm proposed in the section 5.4. Another limitation is that the local ordering of operations is fixed after initial simplification. While this is necessary for animations, a partial re-simplification

might be desirable after huge deformations.

Like all multi resolution modeling techniques, the proposed algorithm is limited to geometric modifications of the vertices. Modifying the connectivity of the mesh would require rebuilding of the progressive mesh data structure. While it would be possible to locally re-compute the operations after large deformations, changing the mesh connectivity would require longer processing times than geometric modifications.

CHAPTER 7

Procedural Mesh Generation

Movies, simulations, and computer games allow to explore a wide variety of realistic, fictional terrains. In some cases, using real terrain would break the illusion of exploring unknown planets. The computer games Spore and Civilization employ procedural models to generate planets from a set of rules automatically. Procedural models have the advantage that no geometry needs to be stored or streamed, instead they are created when needed. While procedural models may use numerous parameters, default values and help texts allow the user to tweak the planets as desired with minimal effort. In recent years, these algorithms were improved to interactively adapt the geometry to a moving camera, in order to support view-dependent level of detail.

While terrain can be generated quickly using previous procedural models, these terrains lack realistic rivers. Rivers are vital for life and can be important for navigation, as rivers often lead to communities or to the sea. Erosion simulations model the natural processes that form rivers. Unfortunately, these algorithms are computationally expensive and can therefore not be used to generate a locally adaptive, high resolution landscape during a fly-through. Instead of attempting to recreate the physical processes of erosion, the proposed algorithm for river networks that obey the following observations:

- While endorheic basins exist, most continental areas transport precipitation to the sea over the river networks.
- River networks are surrounded by valleys between mountains and hills.
- Rivers do not cross, they are mostly above ground, and they follow the steepest decline until they reach the coast.

I call such river networks realistic. The proposed procedural algorithm creates complete planets and landscapes with realistic river networks without performing an erosion simulation, rendering it suitable for on-the-fly generation of terrain. The algorithm starts by creating a coarse representation of the terrain. Additional geometry has to be produced in order to locally adapt the geometry to the camera position and perspective as the

user traverses the terrain. This must be done in a manner that is consistent with the constraints named above. The algorithm uses massively parallel graphics hardware to reach sufficient performance. Figure 7.1 shows a zoom in on the planet generated with the algorithm.

In summary, the main contribution is a novel algorithm that combines the following properties:

- **Adaptive level of detail:** The terrain metadata are stored in the edges and vertices of a mesh and describe rules to refine terrain based on that information. As a result, the algorithm generates river networks at adaptive level of detail.
- **Realistic river networks without an erosion simulation:** Water levels are computed directly and rivers carve into the landscape without an iterative simulation of water movement. Plausibility of the river networks is maintained while adapting the level of detail.
- **Fast terrain synthesis:** The data structures support massively parallel operations, allowing us to generate a base mesh in less than a second and to process refinements in real time.
- **Reproducibility:** The results of the algorithm are reproducible. This is necessary to ensure that the same terrain is generated when the player returns. It would also

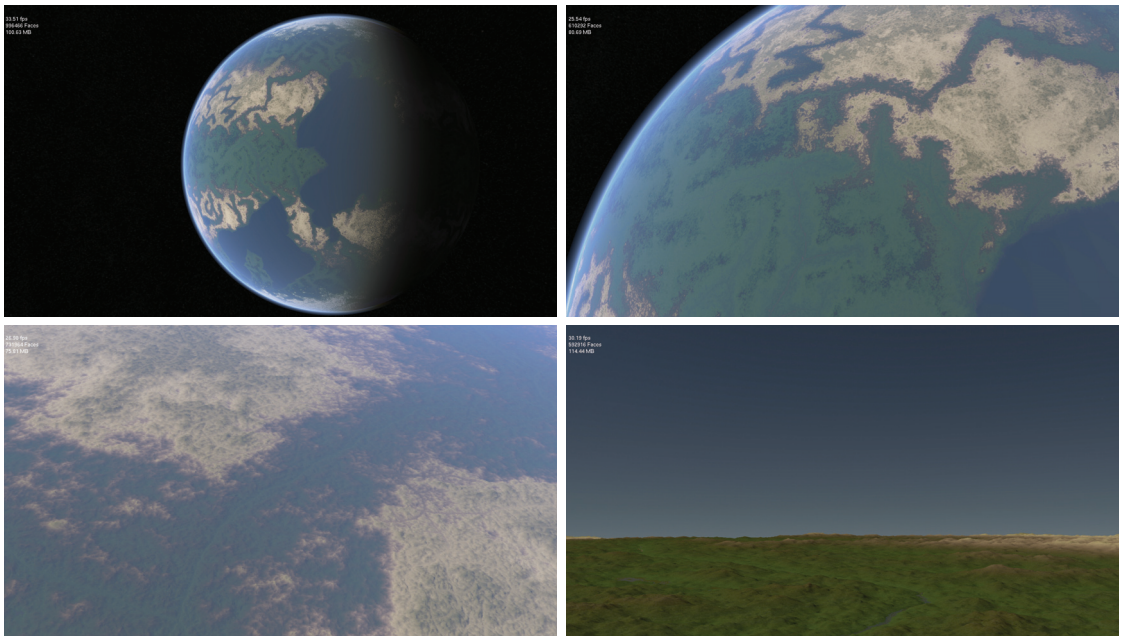

Figure 7.1: As the user zooms in, terrain geometry is created to adaptively refine the planet.

guarantee that all players see the same terrain if the algorithm would be integrated into a networked game.

The remainder of this chapter is structured as follows: Section 7.1 analyzes why previous systems cannot generate terrain with river systems spontaneously. Section 7.2.1 gives an overview of the operations, reproducibility and computing water levels. Section 7.2.2 describes generating the base mesh in detail. In Section 7.2.3, the adaption algorithm for real-time rendering is outlined. Finally, the algorithm is evaluated in Section 7.2.4, which leads to the conclusion (Section 7.2.5).

7.1 PreviousWork

Geomorphology studies the processes that shape the relief of Earth. Among these are crust movements, vulcanism and erosion. Erosion can be caused by temperature changes (thermal erosion), water (fluvial erosion), glaciers (glacial erosion), wind (eolian erosion), and other effects.

7.1.1 Fractal and Procedural Approaches

Mandelbrot [Man83] has analyzed the fractal nature of many types of objects that are used in this thesis, including mountains, river networks, and coast lines. Fournier et al. [FFC82] demonstrated how to produce terrain or entire planets using midpoint displacement. Their algorithm starts with a polygon or sphere and recursively inserts new vertices which split polygons into several new polygons. A new vertex's altitude is the average altitude of the surrounding vertices, plus or minus a random value depending on the length of the edges. Midpoint displacement can be used to increase the resolution of terrain. Bokeloh and Wand [BW06] propose a GPU-based implementation of the midpoint displacement algorithm.

Kelley et al. [KMN88] proposed an algorithm for generating procedural terrain with river networks. Their algorithm uses the observation that water flows along edges that are lower than the surrounding landscape. They first create a river network, calculate river vertex altitudes, and finally assign higher altitudes to the surrounding mountains. All tributaries in the terrain lead into a single, main river.

Hnaidi et al. have proposed a method that defines terrain from user-defined control curves that include ridge lines, river beds, hills, cracks and cliffs [HGA*10]. Two-dimensional piecewise Bezier cubic splines are used to define the control curves. The terrain is computed from a set of partial differential equations which are solved on the GPU.

Several algorithms that generate new terrain by transplanting detail from existing terrain have been proposed [BSS06, EF01, ZSTR07]. A number of approaches generate terrain that satisfies user-given constraints [ST89, PGTG04, SS05, Bel07]. These algorithms could also be used for procedural modeling by defining the constraints procedurally. Prusinkiewicz

and Hammel integrated midpoint displacement and the creation of a single river into a text rewriting system [PH93]. While a lot of research has focused on modeling streets using grammars, most recent approaches model streets as splines or graphs without grammars [GPMG10, LSWW11].

7.1.2 Erosion Simulation

Musgrave et al. proposed to simulate fluvial and thermal erosion on fractal terrain [MKM89]. During every step of erosion simulation, rain is dropped onto the surface and gathered in lakes until it leaks at the lake's lowest border and forms a river bed. As the simulation proceeds, many initial lakes are converted to river beds in this manner and a river network is created. Numerous extensions to erosion simulation have been proposed. Beneš and Forsbach proposed a data structure that stores several layers of material in a grid. For each layer, material information and thickness are stored [BF01a]. They also showed that if the terrain is split into strips, erosion simulation can run in parallel for each strip, only the boundary areas need to be treated separately [BF01b] and they demonstrated how to integrate evaporation into erosion simulations [BF02]. Rather than calculating erosion based on amounts of exchanged water, erosion can be simulated with particles [CMF97, KBKv09]. An optimized GPU implementation of erosion simulation has been proposed by Št'ava et al. [SBBK08].

7.1.3 Terrain Rendering and Parallel Level-of-Detail

View-dependent simplification has been an active field of research over the last two decades. Hoppe [Hop96] introduced progressive meshes that smoothly interpolate between different levels of detail. Depending on the view position and distance, a sequence of split- or collapse operations is performed for the vertices to generate a view-dependent simplification. The inter-dependency of split operations can either be encoded explicitly [XV96] or implicitly [Hop97]. Hoppe later optimized the data structures and improved the performance of the refinement algorithm [Hop98]. Duchaineau et al. store triangles in a binary tree, where each level stores the geometry for a single level of detail [DWS*97]. Pajarola and DeCoro [Paj01, PD04] developed an optimized sequential view-dependent refinement algorithm. Losasso introduced the geometry clipmap, which stores geometry for quadratic regions centered around the user, similar to mipmapping [Los04]. Hu et al. [HSH09] proposed a parallel adaption algorithm for progressive meshes. They introduced a relatively compact explicit dependency structure that allows to group vertex splits and half-edge collapses into parallel steps. The drawbacks of this technique are that the explicit dependencies need additional memory and that only half-edge collapses are supported. A more compact data structure for progressive meshes was proposed by me at chapter 5. It is based on Hoppe's original view-dependent refinement algorithm [Hop96] and supports a massively parallel adaption algorithm.

7.1.4 Analysis

Table 7.1 summarizes the features of the previous algorithms. Ideally, such an algorithm would be interactive, support realistic water and erosion effects, and immediately generate terrain at any desired level of detail while the user explores the terrain. For a number of algorithms, the level of detail is fixed. In other cases, the level of detail is limited by the functions that are used to define the terrain. At high polygon counts, additional polygons make features rounder but do not add further detail. In these cases, Table 1 reports the level of detail as limited. Midpoint displacement can be used to add further detail to a terrain [Bel07]. Unfortunately, the algorithm lacks the necessary rules to prevent introducing mountain peaks into rivers. Some algorithms do not support river networks. However, water effects are vitally important as river networks are a defining element of natural landscapes. As a result, in related work, either the level of detail is limited or river networks are missing.

Both midpoint displacement and the method of Kelley et al. come very close to satisfying the stated requirements. The one lacks river networks, while the other lacks adaptive level of detail. Both methods also operate on a mesh. This thesis demonstrates how they can be combined into a new algorithm. A parallel implementation is required to reach sufficient performance for interactive applications.

Algorithm/Authors	**Input**	**Output**	**Precomputation/Size**	**Water Effects**	**Level of Detail**
Midpoint Displacement	Parameters	Mesh	Very fast	Global sea level	Adaptive
Kelley et al.	Parameters	Mesh	Very fast	River networks	Fixed
Erosion Simulation	Grid	Layered grid[BF01a]	Minutes	Many	Fixed
Constrained Modeling	Constraints	Grid	2.97s / 1024×1024 [Bel07]	Global Sea Level	Limited
Transplant Terrain	Two grids	Grid	2.5 s / 2794×394 [BSS06]	Global Sea Level	Fixed
Hnaidi et al.	Control curves	Grid	0.3 s / 1024×1024 [HGA*10]	User-defined rivers	Limited
Proposed algorithm	Parameters	Mesh	Very Fast (Table 7.4)	Realistic river networks	Adaptive (Fig. 7.6)

Table 7.1: Comparison of features in previous algorithms.

7.2 River Networks for Instant Procedural Planets[39]

The proposed algorithm consists of two phases. In the first phase, the algorithm creates a rough representation of the planet, the so-called base mesh: It creates a sphere, assigns continents, river networks, and altitudes (Section 7.2.2). The second phase is adaption (Section 7.2.3), where the algorithm interactively and adaptively refines the terrain while the user moves about freely. The adaption phase consists of a number of steps that were optimized to take advantage of massively parallel graphics hardware.

39 In Computer Graphics Forum (PG2011) [DGGK11].

7.2.1 Overview

The mesh data structure is used because the algorithm of Kelley et al. requires labeled edges. While it would be possible to store the planet in a displacement map wrapped around a sphere, only eight directions are possible for transporting water between neighboring cells, and a solution would be needed that can exceed these 8 directions when zooming in. Otherwise, parallel rivers would emerge. The mesh data structure supports two atomic operations – edge split and vertex collapse – that are used to manage the level of detail. Edge split operations can be applied to increase the level of detail locally when the user comes closer to parts of the terrain. The reverse operation, vertex collapse, restores the representation at the lower level of detail. Figure 7.2 shows a split operation that creates faces f_3 and f_4, edges e_2, e_3, e_4 and vertex v_{nv}, while a vertex collapse operation reverses this operation by removing these entities. The designations in Figure 7.2 are used in the entire chapter.

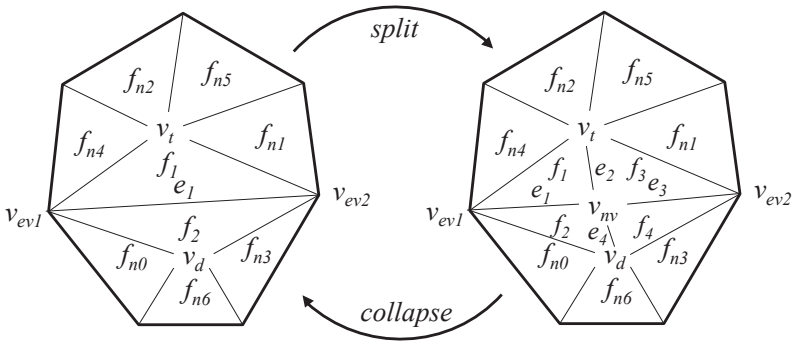

Figure 7.2: Vertex collapse and edge split operation.

7.2.1.1 Reproducibility

Procedural models can often be recreated from a random seed, but a naïve implementation of midpoint displacement would still produce different results when zooming into the terrain along a different path. This is because the adaptive refinements would apply the random values in different local order. However, when the terrain is reproducible, only its current representation needs to be stored, as geometry at different levels of detail can be created when needed. Otherwise, exploring the terrain would create new data until storage is exhausted. Therefore, the method asserts that the same terrain is generated regardless of the path of exploration, which allows several players to explore a planet without streaming geometry over networks. A single random seed is used to compute the entire planet. The base mesh is produced from the random seed using a deterministic algorithm and is thus reproducible, including the random seeds for all vertices. As the order of refinements may vary, reproducibility of the edge splits is guaranteed using the random seeds stored in the vertices. The sum of the seeds of vertices v_{ev1} and v_{ev2} is used

as the seed of v_{nv}. However, the pseudo-random numbers alone are not enough to ensure that refinements are reproducible. The algorithm also need to preserve the local order of the operations, so a split level is assigned to every edge. For each split, the edge's split level is incremented and decremented it for each collapse. Similar to [ESV99], only the edge with the lowest split level can be split in each face.

7.2.1.2 Types of Edges and Faces

A flag in each face stores whether the polygon belongs to the sea or to a continent. Edges are marked as sea, coast, or continent depending on their surrounding polygons. In addition, edges between continental polygons may be flagged as rivers. If a river edge is split, two river edges and two continent edges are created. When a coast edge is split, there will be two coast edges, one sea edge, and one continent or river edge. If a sea edge is split, there will be four new sea edges. When a continent edge is split, the new vertex v_{nv} will have four new continent edges. If the new continent faces are not connected to the river network, the edge between the new vertex v_{nv} and an existing river vertex is converted to a river, to assert that polygons created later are still connected to the river network. While an edge only has one type, vertices have all the types of their incident edges.

7.2.1.3 Water Levels

The ground and water altitudes is stored in each vertex to define water levels for the sea and rivers. If the altitude of the water surface is higher than the ground altitude, the vertex is submerged. If the ground altitude of the river vertex is higher than the water altitude, the vertex is not submerged, but its edges may still be partially submerged by adjacent submerged vertices. While executing a split operation, the ground altitude and the water altitude of v_{nv} are calculated from the values of the adjacent vertices v_t, v_d, v_{ev1} and v_{ev2}. As a result, polygons that have both submerged vertices and vertices above water level are partially flooded and form coasts and river banks. A 2D lookup texture is used to define colors for the climate zones and water depth. The ground altitude, water altitude and geographical position of the vertex are used to compute the texture coordinates. A high quality shader is used to create the water effects.

7.2.2 Planet Generation

Base mesh creation must be fast as the user wants to start exploring the planet without delay but still all information for refining the geometry has to be generated. This process consists of two steps: First, a base shape and the land masses are generated. Then the inital river networks is produced.

7.2.2.1 Base Shape and Continents

The base shape is created by inserting vertices into an octahedron to form a sphere. During each split, each new vertex $\mathbf{v_{nv}}$ needs to be lifted to the surface of the base shape:

$$\mathbf{v_{nv}} \leftarrow (r + a_{nv}) \frac{\mathbf{v_{nv}} - \mathbf{c}}{\|\mathbf{v_{nv}} - \mathbf{c}\|} + \mathbf{c}, \tag{7.1}$$

where r is the sphere's radius, a_{nv} is the vertex's ground altitude, and $\mathbf{c}$ is the sphere's center. Positions for new vertices $\mathbf{v_{nv}}$ are chosen from a randomized weighted sum of the surrounding vertices:

$$\mathbf{v_{nv}} = (1 - \eta)\left(\xi v_{ev1} + (1 - \xi) v_t\right) + \eta(\xi v_d + (1 - \xi) v_{ev2}), \tag{7.2}$$

where ξ,$\eta \in [0.25,0.75[$ are uniformly distributed pseudo-random numbers.

In this context, a continent is a connected land mass above sea level. Initially, all faces are labeled as sea. For every continent, a starting face is selected and labeled as a continent. Faces that have at most one pure sea vertex can be added to the continent. The other vertices in a new face must already belong to the continent. This ensures that two continents are always separated by an edge. The face and its edges are labeled to belong to the continent. This is repeated until the percentage of the total land mass reaches a user-defined value. Any edges and vertices between continental and sea faces are marked as coast. Alternatively, instead of creating a random base mesh, types of polygons, edges and vertices are read from a digital elevation model.

Ground altitudes for continental and coastal vertices are assigned during river network creation and are initialized with zero. Sea vertices are assigned an altitude below sea level and a water altitude that equals the sea level, ε_{sea}. If terrain below sea level is also required, pure midpoint displacement can be used to compute altitudes. In order to generate pseudo random numbers for the vertices in a reproducible manner, each vertex is assigned an initial random seed and its split count is set to zero.

7.2.2.2 Initial River Networks

At this point, continents consist only of continent and coast edges and vertices. The algorithm still need to generate river networks and compute continental vertex altitudes. First the maximal depth of the rivers $\epsilon_{river} < \epsilon_{sea}$ is defined. Creating the river networks starts at the river mouths. The algorithm consider all vertices in pseudo-random order, looking for continental vertices that are adjacent to a coast vertex. In a river mouth, typically only one river mouths into the sea, therefore the coast vertex should not have a river edge yet. The edge between the chosen vertices is flagged as a river edge. In order to complete the river networks, the algorithm pick edges that connect a river vertex with a continent vertex in pseudo-random order and convert these edges to river edges. Two

rivers may merge in a river vertex, but if possible, alternative river edges should be used to prevent merging more than two rivers in a single vertex. When all continental vertices have been connected to the river network, the river networks are complete.

While the river network is created, ground altitudes and water altitudes are assigned to the river vertices, starting from the coast vertices at sea level:

$$a_v = a_u + e_a l_e \xi, \;\; e_a = \frac{a_{max\_river}}{l_r}, \tag{7.3}$$

$$w_v = a_v + e_w l_e, \;\; e_w = \frac{\varepsilon_{river}}{l_{cr}}, \tag{7.4}$$

where v is the current vertex, u is reached by v's outgoing river edge, a_u, a_v are the ground altitudes, w_v is the water altitude of v, average ground elevation e_a, average water elevation e_w, river length l_r, length of the river between v and river spring l_{cr}, length of the current edge l_e and $\xi \in [0,1[$ is a uniformly distributed pseudo-random number. Assigning river altitudes using a constant elevation leads to sharp cliffs in places where a branch of a long river neighbors a branch of a smaller river. Instead, the algorithm assign a maximum altitude a_{max_river} for the river spring, depending on the length of the river. The ground altitude of the river mouth is $\epsilon_{sea} - \epsilon_{river}$. The ground altitudes of the river springs are not allowed to exceed a_{max_river} and the water altitude is equal the ground altitude. The altitude for all other river vertices is interpolated linearly between these (see Figure 7.3). Vertex normals are stored for lighting. As river vertices are hidden, this variable is used to store a tangent towards the river mouth instead, which is used to generate round rivers during later vertex splits.

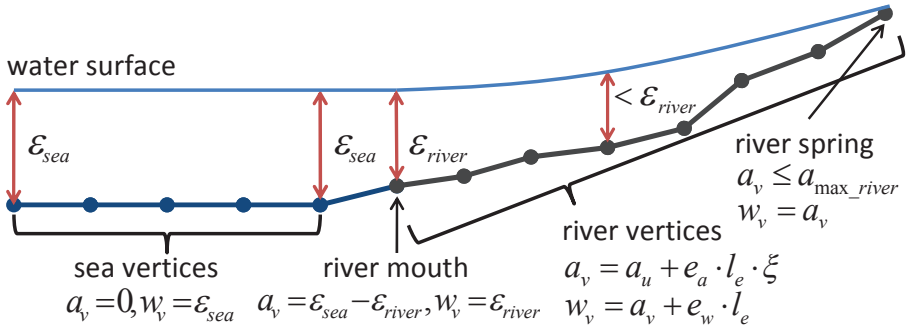

Figure 7.3: Water and ground altitudes of the river vertices.

Now, the river networks are separated by continent or coast edges but there are no river beds. Therefore, the algorithm insert a continent vertex into every continent edge between two river vertices or between a river vertex and a coast vertex. Coast edges with two river mouth vertices must be split in the same manner. While producing the river networks, pure mountain vertices are inserted to separate the rivers. These vertices are placed at higher altitude than their surrounding river vertices, so the algorithm compute an altitude

a_{nv} for $\mathbf{v_{nv}}$ using the altitude a_r of the highest adjacent river vertex $\mathbf{v_r}$:

$$a_{nv} = a_r + e_m l_e \xi, \tag{7.5}$$

where e_m is the elevation of mountain edges, l_e is the horizontal length of the edge between $\mathbf{v_{nv}}$ and $\mathbf{v_r}$, and $\xi \in [0,1[$ is a uniformly distributed random number. This ensures that rivers follow the steepest decline, because the rivers are surrounded by continental vertices at higher altitude. Table 7.2 gives an overview of the parameters used in the algorithm.

Symbol/Name	Description	Value
Basic shape	Sphere, ring or flat polygons	Sphere
r	Size or radius	6371 km
Continents	Number of continents (optionally islands)	6
Land	Percentage of land area to total planet's surface	50%
Bitmap	Alternatively, to define the continents	
Edge length	Minimum edge length	1 cm
Base shape accuracy	Number of triangles for base shape	5000
2D Lookup Texture	To define colors for the climate zones	
a_{max_river}	Max. river altitude	12 km
$a_{max\_mountain}$	Max. mountain altitude	13 km
ε_{sea}	Global water level (sea level)	3 km
ε_{river}	Max. river depth	300 m
l_{rmin}	Min. river length	200 km
s_r	Min. river slope	1 m/km
s_{rb}	Min. riverbed slope	300 m/km
s_p	Min. slope near river	1 m/km
ε_{rb}	Riverbed edge height above water	10 m
s_g	Max. ground slope	1000 m/km

Table 7.2: Main parameters used to define a planet in the algorithm.

7.2.3 Runtime Algorithm

The adaption algorithm is divided into several consecutive steps to take advantage of massively parallel hardware. The partitioning is required for thread synchronization while each step can be processed completely in parallel. First, the algorithm test which edges have to be split and which vertices must be collapsed to adapt the mesh to the new camera position. Then the selected operations are performed. This mesh is then used as input for the next frame to exploit temporal coherence.

The main data structures required for rendering are the vertex buffer containing the position and normals and the index buffer storing the connectivity of the adapted mesh. Both are stored as vertex buffer objects (VBOs) and are therefore separate from all other data. Table 7.3 gives an overview of the dynamic data structures that are discussed in the following.

Buffers	Elements	Memory (bytes per entity)	New entities (per new vertex)
active edges	face ID (×2)	8	3
	length	4	
	split count	2	
	type	1	
active faces	index VBO	12	2
	edge ID (×3)	12	
	normal	12	
	type	1	
active vertices	vertex VBO	24	1
	ground altitude	4	
	water altitude	4	
	edge ID	4	
	maximal edge length	4	
	seed	4	
	type	1	
	coast marker	1	
temporary	split flag	4	3
	collapse flag	4	1
	temp	4	3
Total memory (per vertex)			193 byte

Table 7.3: Elements of the data structure.

7.2.3.1 Vertex State Update

In the first step, the necessary operations are determined. If the vertex v needs to be split according to its refinement criteria, the *split* flag is set in its state. Otherwise, the *collapse* flag is set if the refinement criteria allow a collapse. In all other cases no operation is required for v. Then the state of the edges is determined by traversing all edges. An edge is split if one of the edge vertices meets the criteria for a split. As some splits and collapses cannot be executed immediately, an additional check has to be performed to remove conflicting operations. For the split operations, only the edge with the lowest split level can be split in each face f. In addition, if any edge is marked for splitting, no vertex of f can be collapsed. If no edge of the face needs to be split, the collapse operations are checked. v_{nv} can only be collapsed if all incident edges (e_1, e_2, e_3 and e_4) have the same split level. This assures that only one edge in a face can be split and only one vertex can be collapsed to avoid race conditions. Split and collapse operations can be executed in parallel provided that each triangle is affected by only one operation. If there are several operations affecting a single triangle, the most important operation is executed immediately, while the other operations may be executed in one of the following frames. Due to camera movement, the priority of delayed operations may change. Splitting all three edges on a triangle would take three frames.

The algorithm check several criteria to remove invisible vertices. The most simple one

is view frustum culling: A vertex can be collapsed if it lies outside the view frustum regardless of the screen space error. To prevent foldovers and popping artifacts when rotating or panning, the algorithm do however not simply collapse all vertices that are outside of the view frustum but modify the distance d of these vertices for the following LOD selection:

$$\tilde{d} = \left(c_{LOD} \left(\frac{\max(|x|, |y|, |z|)}{w} - 1 \right)^2 + 1 \right) d, \tag{7.6}$$

where x, y, z and w are the homogeneous coordinates of the vertex after projective transformation and c_{LOD} is a constant value. In the experiments, $c_{LOD} = 20$ resulted in a smooth LOD falloff outside the view frustum. Then backface culling is performed and the screen space error evaluated. For splitting and merging vertices based on camera distance, $\frac{l}{\tilde{d}} > c$ is tested, where l is the maximal length of the edge assigned to the vertex (stored) and c is a constant value, and set *split*/*collapse* flags according to the result of the test. Different values of c are used for sea and continent vertices because the sea can be rendered at a lower resolution to reach the same quality as the land. Algorithm 20 summarizes the vertex update operations.

```
foreach vertex v in parallel do
    if need_split(v)
        mark(v, split)
    elif may_collapse(v)
        mark(v, collapse)
foreach edge e in parallel do
    determine_edges_state(e,v_ev1,v_ev2)
    if edge_marked(e, split)
        level_min = get_min_active_split_level(f)
        unmark_dependent_splits(f, level_min)
        unmark_all_collapses(f)
foreach vertex v in parallel do
    if any_vertex_marked(v, collapse)
        if neighboring_edges_level_not_equals(v_nv, e1-e4)
            unmark_collapse(v_nv)
```

Algorithm 20: The parallel vertex states update.

7.2.3.2 Memory Management

Before split and collapse operations can be performed, the algorithm may need to adjust the size of the buffers. The algorithm always reserve slightly more memory than currently required to reduce the runtime cost for allocating memory and copying data when the size of an array is modified. If the size of the vertex, edge or face buffers is too small or significantly too large, new buffers are allocated and the content of the old ones is copied into them. When a reallocation is performed, the buffer size is set to the number of faces

n_f plus a user-defined threshold n_{alloc}. If the buffer is larger than $n_f + 2n_{alloc}$ it is reduced to $n_f + n_{alloc}$.

7.2.3.3 Parallel Edge Splits

After updating the state of all active edges and removing illegal splits and collapses, the operations can be applied. First the splits [SHZO07] are compacted to ensure that each thread performs an operation to improve the thread utilization. Then, the following steps are performed for each split operation (summarized in Algorithm 21):

1. Calculate the seed $s_{nv} = s_{ev1} + s_{ev2}$ of the new vertex v_{nv}, where s_{nv} is the seed of vertex v_{nv}.

2. Two new faces f_3 and f_4 and three new edges e_2, e_3 and e_4 are generated and added to the buffers (Figure 7.2).

3. Change the connectivity of neighboring faces edges and vertices, as demonstrated in Figure 7.2.

4. Assign types to the new faces, edges and vertices using the rules from section 7.2.1.2. The type of f_1 is assigned to the new face f_3 and new edge e_2. Similarly, the type of f_4 and e_4 is the type of f_2.

5. The position of the sea vertices is the center of the edge that is split. For continent and coast vertices the position of v_{nv} is calculated from the adjacent vertices v_{ev1}, v_{ev2}, v_t, and v_d using equation 7.2. The position of the river vertices is calculated from v_t and v_d only:

$$\mathbf{v_{nv}} = (1 - \xi)v_t + \xi v_d, \tag{7.7}$$

   where again $\xi \in [0.25,0.75[$ is a uniformly distributed pseudo-random number, calculated with a seed of v_{nv}. In addition, ξ is biased to generate smooth rivers. If the edge length is less than the minimal river length l_{rmin} the use:

$$\xi' = \xi + \frac{(v_s - v_d) \cdot (v_t - v_d)}{\|v_t - v_d\|^2}, \tag{7.8}$$

   with

$$v_s = \frac{v_{ev1} + v_{ev2}}{2} + \|v_{ev1} - v_{ev2}\| \frac{t_{ev1} - t_{ev2}}{4}, \tag{7.9}$$

   where t_{ev1} and t_{ev2} are the stored tangents of v_{ev1} and v_{ev2}. Equation 7.9 assumes that the river flows from v_{ev1} to v_{ev2}. Additionally, the tangent is calculated for the new river vertex v_{nv} and stored instead of the normal. Finally, v_{nv} is added to the vertex buffer.

6. Calculate the ground altitude a_{nv} and water altitude w_{nv} of v_{nv}. The altitude of sea vertices is simply the sea bottom and the water altitude the sea level. For river vertices it is $\frac{1}{2}(w_{ev1} + w_{ev2})$ and $\frac{1}{4}(w_{ev1} + w_{ev2} + w_{vt} + w_{vd})$ for all others. For continent vertices the algorithm then check if it can construct a new river arm, where w_{nv} is set to a_{nv}. It can be generated if the split edge is not a river edge and one of the four edges can be converted into a river. This is only possible if the edge is longer than l_{rmin} and a_{nv} can be at least $l_e s_r$ above the other river vertex and below all neighboring non-river vertices c:

$$a_{nv} < \min_c \left(a_c - \min\left(l_e s_{rb}, \varepsilon_{rb} + (l_e s_p),\right)\right). \tag{7.10}$$

where l_e length of the edge between v_{nv} and neighboring non-river vertex v_c, s_r minimal river slope, s_{rb} minimal riverbed slope, s_p minimal slope near river and ε_{rb} riverbed edge height above water. For continent vertices similar bounds apply (see Figure 7.4):

$$a_{nv} > \min_{bw} \left(\min\left(a_{bw} + l_e s_{rb}, w_{bw}\varepsilon_{rb} + (l_e s_p)\right)\right), \tag{7.11}$$

$$\max_c \left(a_c - (l_e s_g)\right) < a_{nv} < \min_c \left(a_c + (l_e s_g)\right), \tag{7.12}$$

where s_g maximal ground slope, bw are the neighboring river vertices or those covered by water and c the remaining ones. The final altitude is then a random value inside the previously computed bounds.

7. Calculate normals for f_1, f_2, f_3 and f_4 and the vertex normal of v_{nv} unless it is a river vertex. The normals for each vertex are computed from the surrounding face normals.

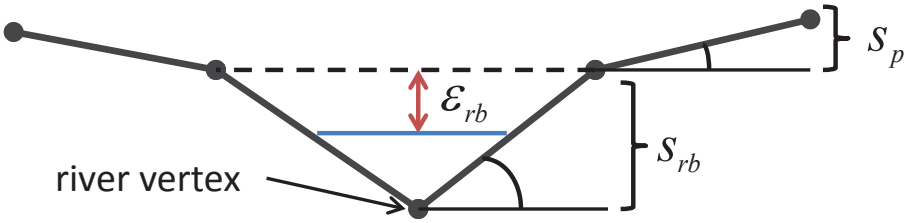

Figure 7.4: Riverbed shaping parameters.

7.2.3.4 Parallel Vertex Collapses

After applying the split operations, the collapse operations need to be performed. Each collapse operation removes vertex v_{nv} and edges e_2, e_3 and e_4. In addition, faces f_3 and f_4 degenerate and are removed from the mesh. Then faces f_1 and f_2, edge e_1, and all incident faces and edges are relinked (Figure 7.2). The member variable `edgeID` is used in each vertex to store an edge that contains the vertex. This helps us locate the other

```
compact(splits)
foreach split edge e in parallel do
    calc_seed(v_nv)
    add_new_faces_edges_vertex(e1-e4, f1-f4, v_nv)
    relink_neighbors()
    assign_types(e1-e4, f1-f4, v_nv)
    calculate_altitudes(v_nv)
    v_nv = split_edge(e)
    calculate_faces_normals(f1-f4)
```

Algorithm 21: Parallel edge split algorithm.

edges around the vertex quickly for collapse operations. When all operations have been applied, the maximal length of the adjacent edge of the new vertices and the vertices in the neighborhood of the split and collapse operations need to be recalculated. Algorithm 22 shows the parallel processing of the edge collapse operations and how maximal lengths are recalculated.

```
foreach vertex v in parallel do
    if marked(v, collapse)
        remove_faces_edges_vertex(v)
        relink_neighbors(v)
foreach vertex v in parallel do
    if required_length_recalc(v)
        recalc_max_length(v)
```

Algorithm 22: Parallel edge collapse algorithm.

7.2.3.5 Buffer Compaction

If vertices were removed, the active vertices (including the vertex VBO), active faces (including the index VBO), and active edges are compacted in the final step of adaption. Note that when compacting the vertices, faces or edges, the references to them must be updated accordingly. While the compaction of the faces and thus the indices is mandatory since the index VBO are used for rendering, the compaction of the vertices and edges is not. The latter two only need to be compacted every few frames to prevent bloating of the buffers. To save time and memory, a specialized in-place compaction algorithm (Section 5.2) is used since the ordering does not need to be preserved.

7.2.4 Results

The test system consists of a 3.333 GHz Intel Core i7-980X CPU with 6 GB of DDR3-1333 main memory, 16 lanes PCIe 2.0 slot, and a GeForce GTX 580 (841/4200MHz). OpenGL is used for rendering and CUDA for the adaption algorithm. Terrains were reproducible over networks and on different PCs. All images were generated under real-time conditions

at a resolution of 1920×1080 with an edge length of 0.5 pixel for the land and 5 pixels for the water. The number of polygons is limited dynamically to a value that ensures that 20 to 30 frames can be rendered per second, to adjust the algorithm to the capabilities of different hardware.

Table 7.4 lists the number of rendered faces, the total time (rendering and adaption per frame), total and adapt number of triangles per second (TPS), and the memory consumption for the views shown in Figure 7.5 and the fly through in the accompanying video, where the numbers are taken from the most complex frame. A base mesh with approximately 5000 faces is used. Creating the base mesh took 0.27 seconds. Table 7.2 shows the values that were used to produce the accompanying results. While the user explores the planet, the dynamic data structures reside on the graphics card only. This has the advantage that the data can be rendered and adapted without passing it over the PCIe bus. The algorithm can process up to 34/100 (total/adapt) M TPS for static views. The high quality shaders are used to demonstrate that the algorithm is suitable for real-time rendering of terrains at high quality. However, the shaders require 50% to 80% of the rendering time. With simpler shaders, the adaption time lies between at 40% to 70% of the total frame time.

model	*# rendered faces*	*memory (MB)*	*frame time (ms)*	*total/adapt M TPS*
orbit view	1,153,435	115.8	35.8 (38.3%)	32.2/84.1
ground view 1	1,296,562	130.2	38.5 (33.6%)	33.7/100.2
ground view 2	858,562	92.6	33.8 (36.9%)	25.4/68.8
video (max.)	1,041,970	139.5	58.4 (46.8%)	17.8/38.1

Table 7.4: Memory consumption, total rendering time and total number of triangles per second (TPS) of the different views. The ratio of adaption time compared to total time is given in parenthesis.

Timings for adaption and rendering together with memory consumption and the number of active faces for the fly through in the accompanying video are shown in Figure 7.6. The data structures consumed less than 139.5 MB and the average frame rate is 30 frames per second (fps). The peak performance for dynamic views is 27/72 MTPS (total/adapt) and up to 7/15 MTPS can be generated. The proposed approach quickly reacts to changes of the view direction with fast adaption of the terrain complexity. Due to the high adaption performance, only few popping artifacts are visible in the video despite the fast movements. Figures 7.1, 7.5, and 7.8 show example planets.

Finally, the runtime of each step is analyzed inside the adaption and rendering algorithm in Figure 7.7. The most expensive step of the algorithm is the state update, because it must be performed for each active vertex. The time spent on mapping and unmapping the index and vertex buffers for access by CUDA/OpenGL cannot be reduced or prevented. Rendering takes up to 63% of the frame time.

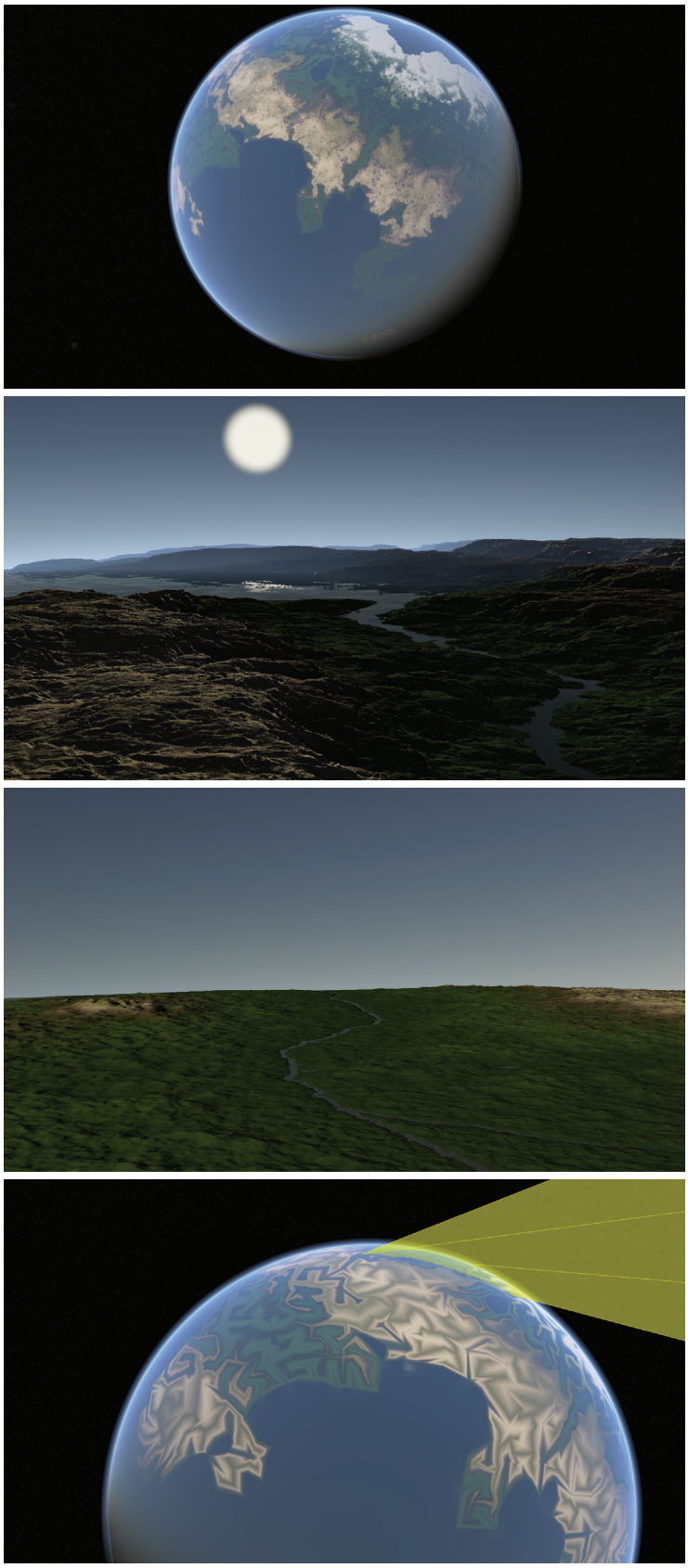

Figure 7.5: The images show the models as rendered from the point of view. The bottom image demonstrates that parts of the terrain that are outside the view frustum (yellow) are reduced to the base mesh.

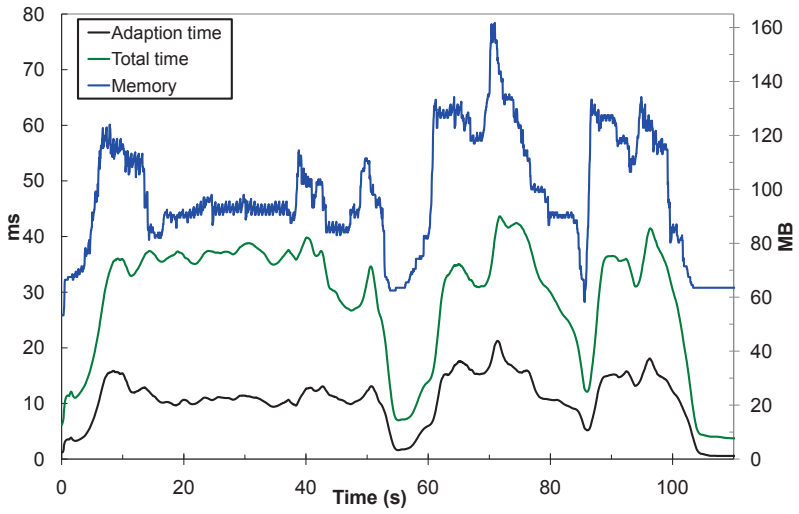

Figure 7.6: Timings and memory consumption for the scene using a pre-recorded camera path.

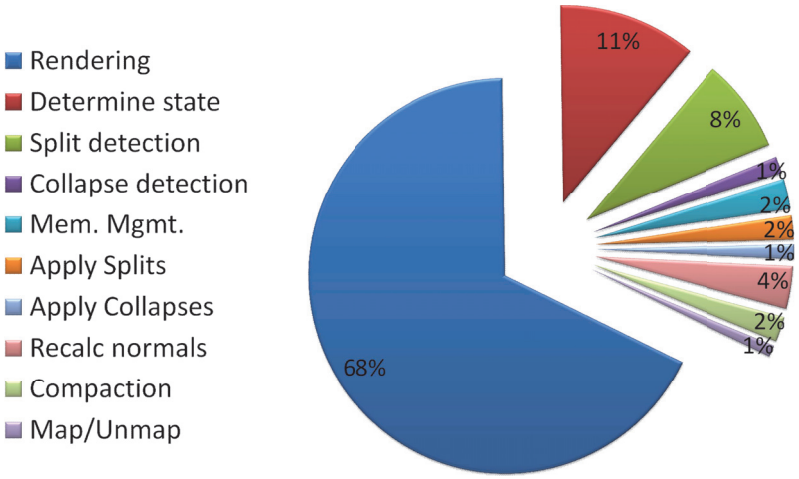

Figure 7.7: Relative time of the several adaption steps compared to rendering.

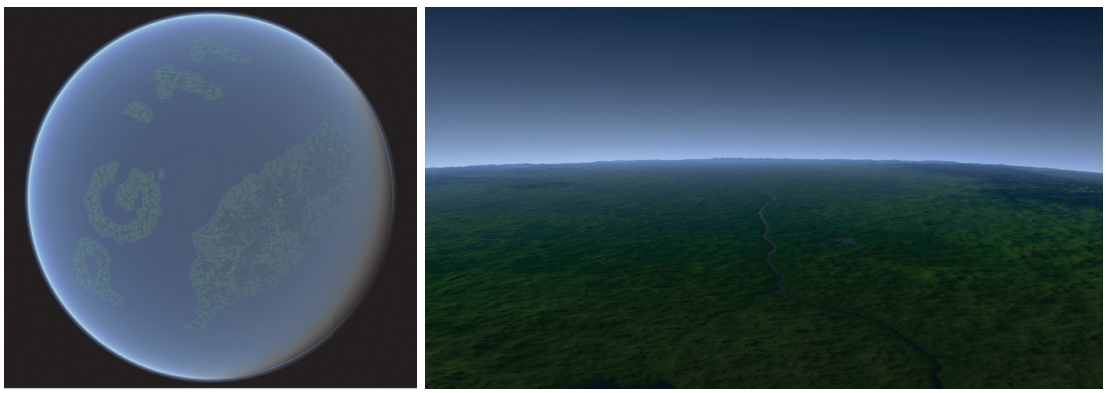

Figure 7.8: Left: Alternatively, the algorithm can be used to generate river networks for user-provided maps. Right: Meandering rivers.

7.2.5 Discussion and Conclusion

In this chapter a new procedural algorithm that spontaneously creates planets or terrain parts with continents and realistic river networks was proposed. It is specifically developed for massively parallel view-dependent adaption. While previous algorithms are not able to generate planets at adaptive level of detail within seconds while ensuring consistency of the river networks, the proposed algorithm is the first to present a parallelized pipeline that combines these features. The algorithm correctly models how valleys and mountain tops differ in that rivers flow through the valleys. While many applications use a fixed level of detail, interactively adapting the level of detail to the camera perspective is a necessity when dealing with large terrains, such as planets. The algorithm does not require storage except for the mesh representing the terrain, and the geometry does not have to be streamed over network, even when several users wish to explore the same terrain. Only a small parameter set and an initial random seed are needed to completely recreate a planet because of reproducibility.

The system by Kelley et al. produces only a single river network, and it does not support refinement [KMN88]. The proposed technique is suitable for interactively adapting large terrains to a moving camera. The only other technique that can do this is the midpoint displacement algorithm by Fournier et al. [FFC82], which does not produce river networks. When using midpoint displacement on realistic terrain, rivers may be interrupted by mountains, which would likely block the rivers, resulting in large endorheic basins or inconsistencies. In contrast to that, the rules implemented by the proposed algorithm ensure consistency of the river networks.

Erosion simulation can also produce eroded terrain with continents and river networks, but that family of algorithms requires much more computation time and has to run supervised. Otherwise, either too many large lakes remain, or the rivers carve too deeply into the terrain. Bad choices for the parameter values often mean that the entire simulation must be restarted, whereas the proposed algorithm produces a viable solution much faster. While it may be possible to combine erosion simulation with adaptive level of detail, ultimately similar problems would have to be solved, and users might note changes in the terrain caused by the simulation. In contrast to that, the terrain generated in the proposed algorithm is immediately realistic and stable.

The proposed system yields correct results for glaciofluvial erosion. Thermal and eolian erosion are modeled by limiting terrain slope. These effects are computed in the proposed method without time-intensive simulations. The proposed algorithm can be used for simulations that require spontaneously created terrain. This includes computer games and learning to steer vehicles, e.g. flight school.

CHAPTER 8

Conclusion and Future Work

In addition to the discussion in the previous chapters, this chapter gives an brief overview of the algorithms proposed in this thesis. Moreover, limitations and possible extensions are discussed.

8.1 Conclusion

Highly detailed models are commonly used in computer games and other interactive rendering applications. The complexity of such models still grows faster than the ability of the graphics hardware to render them in real-time. This thesis addressed this problem by introducing data structures for efficient LOD rendering, simplification, edition and generation of meshes optimized for parallel processing and low memory consumption on the GPU.

In chapter 3 a parallel implementation of the quadric error simplification developed by Garland and Heckbert [GH97a] is proposed. The generated meshes are identical to those produced by the sequential algorithm for a given error bound. On a customer level graphics card, the method can generate a set of LODs for a model containing over 4 million faces in less than a second[40]. This is comparable to loading the generated LODs from disk and significantly faster than network transfer. It is significantly faster compared to existing approaches.

A combination of a parallel out-of-core marching cubes implementation with a parallel stream simplification algorithm is presented in the chapter 4. In contrast to previous approaches the proposed algorithm can generate simplified iso-surface meshes of the same quality as standard simplification after complete iso-surface extraction. Since the presented parallel algorithm does not need to store and simplify the complete mesh, the pre-simplified iso-surfaces of large volume data sets can be parallel generated within less than a minute.

In chapter 5 the novel parallel implementations and data structures for in-core and

40 On NVIDIA GTX 580 GPU.

out-of-core view-dependent progressive meshes are proposed. In section 5.2, a compact in-core data structure for progressive meshes is presented. This in-core data structure is extended in section 5.3 for out-of-core memory management. The main problem of the algorithms proposed in section 5.2 and 5.3 and over previous approaches, are the inter-dependency of split operations. Due to the inter-dependency of split operations, the adaption rate is reduced, leading to visible popping artifacts during fast movements. Section 5.4 proposes a novel in-core and out-of-core dependence free progressive mesh representation. It is specifically designed for massively parallel view-dependent adaption of gigabyte-sized models. In contrast to the previous approaches, no splits need to be postponed as they are waiting for others to be applied before them. It is based on a neighborhood dependency free progressive mesh data structure. Using a per operation compression method, it is suitable for parallel random-access decompression and out-of-core memory management without storing decompressed data. In contrast to existing approaches, this allows rendering of very large models with fast movement nearly without popping artifacts.

Progressive meshes are often employed to improve the rendering performance by reducing the number of rasterized triangles. The classical work flow is to generate a model and then use simplification algorithms to construct the progressive mesh. Thus the whole simplification has to be performed again after editing the model. This does not only require additional processing time but also hinders animations of progressive meshes. Based on this observation a real-time parallel multi resolution modeling algorithm for progressive meshes is proposed in the chapter 6. It can be used for real-time editing and animation of complex progressive meshes. Due to the progressive representation it can intuitively modify the overall shape or small scale details. In contrast to existing approaches, it quickly generates a progressive mesh from a complex triangle model. It is also proposed a massively parallel simplification algorithm that generates all required data structures within a few seconds.

In chapter 7, a new procedural algorithm that spontaneously creates planets or terrain parts with continents and realistic river networks is proposed. It is specifically developed for massively parallel view-dependent adaption. The technique is suitable for interactively adapting large terrains to a moving camera. In contrast to previous algorithms, it combines the generation of planets at adaptive level of detail within seconds while ensuring consistency of the river networks. The algorithm does not require storage except for the mesh representing the terrain, and the geometry does not have to be streamed over network, even when several users wish to explore the same terrain. The algorithm can be used for simulations that require spontaneously created terrain. This includes computer games and learning to steer vehicles, e.g. flight school.

8.2 Future Work

For the mesh simplification proposed in chapter 3 and 4, the out-of-core data structure should to be developed. It allows the simplification of very huge models that exceed the amount of available graphics memory. The main limitation of the algorithms is that the computation of the target placement is rather expensive. With increasing number of attributes, this dominates the total runtime. An acceleration of the computation would significantly increase the performance. Another limitation is that the algorithms do not check for triangle flips during simplification, which can produce visible artifacts. A possible extension of the method would be the addition of vertex pair contractions. These could be integrated by adding an additional set of virtual edges before simplifying the mesh for a level. The maximum vertex distance would then be in the range of the error threshold.

The main limitation of the dependency free parallel progressive meshes algorithm proposed in section 5.4 is that the reorganization of the vertices is rather expensive. An acceleration or prevention of this step would significantly increase the performance. Moreover, increasing the compression rate without limiting random access would increase the performance and reduce the required memory.

The main limitation of the progressive mesh editing algorithm presented in chapter 6 is that the model size is currently limited to a few million triangles. In the future it can be expanded for larger models by using out-of-core or possibly even compression techniques as in the algorithm proposed in section 5.4. Another limitation is that the local ordering of operations is fixed after initial simplification. While this is necessary for animations, a partial re-simplification might be desirable after huge deformations.

For the procedural planet generation proposed in chapter 7, the new texturing techniques need to be developed. By using photo-realistic textures, the quality of the terrain can be improved. Such textures are produced using satellites or aeroplanes and can be used to create a sample dataset. This dataset is then used for texturing of the terrain. Additionally, plate tectonics could be simulated to create a more realistic base mesh, which would improve the quality of the generated terrain.

List of Figures

List of Tables

List of Algorithms

Glossary

1D	One-dimensional space
2D	Two-dimensional space
3D	Three-dimensional space
bpv	bytes per vertex
BWT	Burrows Wheeler Transform
CPU	Central Processing Unit
CT	Computer Tomograph
CUDA	Compute Unified Device Architecture
DAG	Direct Acyclic Graph
DDR SDRAM	Double Data Rate Synchronous Dynamic Random-Access Memory
fBm	Fractional Brownian Motion
FLOP	Floating-Point Operation
FPS	Frames per Second
GPC	Graphics Processor Cluster
GPGPU	General Purpose Compution on Graphics Processing Unit
GPU	Graphics Processing Unit
HDD	Hard Disk Drive
HLOD	Hierarchical Level of Detail
ic	In Core
LOD	Level of Detail
MIMD	Multiple Instruction Multiple Data

MRT	Magnetic Resonance Tomograph
MT	Multi-Triangulations
MTPS	Million Triangles per Second
ooc	Out of Core
PM	Progressive Mesh
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Thread
SM	Streaming Multiprocessor
SP	Streaming Processors
SSE	Streaming SIMD Extensions
TPS	Triangles per Second
uchar	unsigned character
uint	unsigned integer
ushort	unsigned short integer
VBO	Vertex Buffer Object
VFC	View Frustum Culling

Bibliography

[AAR05] Alregib G., Altunbasak Y., Rossignac J.: Error-resilient transmission of 3d models. *ACM Trans. Graph. 24*, 2 (April 2005), 182–208.

[ACSE05] Attali D., Cohen-Steiner D., Edelsbrunner H.: Extraction and simplification of iso-surfaces in tandem. In *Proceedings of the third Eurographics symposium on Geometry processing* (Aire-la-Ville, Switzerland, Switzerland, 2005), SGP '05, Eurographics Association.

[AM04] Anh V. N., Moffat A.: Index compression using fixed binary codewords. In *Proceedings of the 15th Australasian database conference - Volume 27* (2004), ADC '04, Australian Computer Society, Inc., pp. 61–67.

[Bel07] Belhadj F.: Terrain modeling: a constrained fractal model. In *Proc. of AFRIGRAPH 07* (2007), AFRIGRAPH '07, pp. 197–204.

[BF01a] Beneš B., Forsbach R.: Layered data representation for visual simulation of terrain erosion. In *SCCG '01: Proceedings of the 17th Spring conference on Computer graphics* (2001), p. 80.

[BF01b] Beneš B., Forsbach R.: Parallel implementation of terrain erosion applied to the surface of mars. In *Proc. of AFRIGRAPH 01* (2001), pp. 53–57.

[BF02] Beneš B., Forsbach R.: Visual simulation of hydraulic erosion. In *WSCG* (2002), pp. 79–94.

[BGB*05] Borgeat L., Godin G., Blais F., Massicotte P., Lahanier C.: Gold: interactive display of huge colored and textured models. *ACM Trans. Graph. 24*, 3 (2005), 869–877.

[Bro28] Brown R.: *A Brief Account of Microscopical Observations, Made in the Months of June, July and August 1827, on the Particles Contained in the Pollen of Plants and on the General Existence of Active Molecules in Organic and Inorganic Bodies.* Reprinted in Edinburgh New Philos. J. 5, 358 (1928), 1828.

[BSS06] Brosz J., Samavati F. F., Sousa M. C.: Terrain synthesis by-example. In *Proceedings of the first International Conference on Computer Graphics Theory and Applications* (2006).

[BW06] Bokeloh M., Wand M.: Hardware accelerated multi-resolution geometry synthesis. In *Proceedings of the 2006 symposium on Interactive 3D graphics and games* (2006), I3D '06, pp. 191–198.

[CCG*01] Cunniff R., Craighead M., Ginsburg D., Lefebvre K., Licea-Kane B., Triantos N.: *ARB occlusion query.* Tech. rep., NVIDIA and ATI, 2001.

[CGG*04] Cignoni P., Ganovelli F., Gobbetti E., Marton F., Ponchio F., Scopigno R.: Adaptive tetrapuzzles: efficient out-of-core construction and visualization of gigantic multiresolution polygonal models. *ACM Trans. Graph. 23*, 3 (2004), 796–803.

[CH91] Clark J., Holton D. A.: *A First Look at Graph Theory.* World Scientific, 1991.

[CH09] Courbet C., Hudelot C.: Random accessible hierarchical mesh compression for interactive visualization. In *Proceedings of the Symposium on Geometry Processing* (2009), pp. 1311–1318.

[Che95] Chernyaev E. V.: *Marching Cubes 33: Construction of Topologically Correct Isosurfaces.* Tech. rep., Technical Report CERN CN 95-17, 1995.

[CKL*04] Choe S., Kim J., Lee H., Lee S., Seidel H. P.: Mesh compression with random accessibility. In *Israel-Korea Bi-National Conf* (2004), pp. 81–86.

[CKLL09] Choe S., Kim J., Lee H., Lee S.: Random accessible mesh compression using mesh chartification. *IEEE Transactions on Visualization and Computer Graphics 15*, 1 (January 2009), 160–173.

[CMF97] Chiba N., Muraoka K., Fujita K.: An erosion model based on velocity fields for the visual simulation of mountain scenery. *The Journal of Visualization and Computer Animation 9*, 4 (1997), 185 – 194.

[Der09] Derzapf E.: Progressive meshes mit cuda. In *Diploma Thesis* (Dec. 2009), Philipps-Universität Marburg. Fachbereich 12 - Mathematik und Informatik, Arbeitsgruppe Grafik und Multimedia Programmierung.

[DFMP98] De Floriani L., Magillo P., Puppo E.: Efficient implementation of multi-triangulations. In *VIS '98: Proceedings of the conference on Visualization '98* (1998), pp. 43–50.

[DG12] Derzapf E., Guthe M.: Dependency free parallel progressive meshes. *Computer Graphics Forum to appear*, to appear (2012), to appear.

[DGG12] Derzapf E., Grund N., Guthe M.: *Parallel Progressive Mesh Editing.* Tech. rep., Philipps-Universität Marburg, Fachbereich 12 - Mathematik und Informatik, Arbeitsgruppe Grafik und Multimedia Programmierung, 2012.

[DGGK11] Derzapf E., Ganster B., Guthe M., Klein R.: River networks for instant procedural planets. *Computer Graphics Forum 30*, 7 (2011), 2031–2040.

[DGGP05] Diaz-Gutierrez P., Gopi M., Pajarola R.: Hierarchyless simplification, stripification and compression of triangulated two-manifolds. *Computer Graphics Forum 24*, 3 (2005), 457–467.

[DJCM09] Du Z., Jaromersky P., Chiang Y.-J., Memon N.: Out-of-core progressive lossless compression and selective decompression of large triangle meshes. In *Proceedings of the 2009 Data Compression Conference* (Washington, DC, USA, 2009), IEEE Computer Society, pp. 420–429.

[DJG*10] Dupuy G., Jobard B., Guillon S., Keskes N., Komatitsch D.: Parallel extraction and simplification of large isosurfaces using an extended tandem algorithm. *Comput. Aided Des. 42*, 2 (2010), 129–138.

[DMG10a] Derzapf E., Menzel N., Guthe M.: Parallel view-dependent refinement of compact progressive meshes. In *Eurographics Symposium on Parallel Graphics and Visualization* (2010), pp. 53–62.

[DMG10b] Derzapf E., Menzel N., Guthe M.: Parallel view-dependent out-of-core progressive meshes. In *Vision, Modeling, and Visualization* (2010), pp. 25–32.

[DT07] DeCoro C., Tatarchuk N.: Real-time mesh simplification using the gpu. In *I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games* (2007), pp. 161–166.

[DWS*97] Duchaineau M., Wolinsky M., Sigeti D., Miller M., Aldrich C., Mineev-Weinstein M.: Roaming terrain: real-time optimally adapting meshes. In *Proceedings of the 8th Conference on Visualization'97* (1997), pp. 81–88.

[EF01] Efros A. A., Freeman W. T.: Image quilting for texture synthesis and transfer. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (2001), SIGGRAPH '01, pp. 341–346.

[EMB01] Erikson C., Manocha D., Baxter III W. V.: Hlods for faster display of large static and dynamic environments. In *I3D '01: Proceedings of the 2001 symposium on Interactive 3D graphics* (2001), pp. 111–120.

[ESV99] El-Sana J., Varshney A.: Generalized view-dependent simplification. *Computer Graphics Forum 18*, 3 (1999), 83–94.

[FB88] Forsey D. R., Bartels R. H.: Hierarchical b-spline refinement. In *Proceedings of the 15th annual conference on Computer graphics and interactive techniques* (1988), SIGGRAPH '88, ACM, pp. 205–212.

[FFC82] Fournier A., Fussell D., Carpenter L.: Computer rendering of stochastic models. *Communications of the ACM 25*, 6 (1982), 371–384.

[Gar99] Garland M.: Multiresolution modeling: Survey & future opportunities. In *EUROGRAPHICS 1999 - State of The Art Reports (STARs)* (1999), pp. 111–131.

[GBBK04] Guthe M., Borodin P., Balázs Á., Klein R.: Real-time appearance preserving out-of-core rendering with shadows. In *Rendering Techniques 2004 (Proceedings of Eurographics Symposium on Rendering)* (2004), pp. 69–79.

[GBK03] Guthe M., Borodin P., Klein R.: Efficient view-dependent out-of-core visualization. In *Proceedings of the 4th International Conference on Virtual Reality and its Applications in Industry (VRAI '2003)* (2003), pp. 428–438.

[GDG11] Grund N., Derzapf E., Guthe M.: Instant level-of-detail. In *Vision, Modeling, and Visualization (VMV2011)* (2011), pp. 293–299.

[GH97a] Garland M., Heckbert P. S.: Surface simplification using quadric error metrics. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques* (1997), SIGGRAPH '97, pp. 209–216.

[GH97b] Garland M., Heckbert P. S.: Surface simplification using quadric error metrics. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques* (1997), pp. 209–216.

[GH98] Garland M., Heckbert P. S.: Simplifying surfaces with color and texture using quadric error metrics. In *Proceedings of the conference on Visualization '98* (1998), pp. 263–269.

[GM05] Gobbetti E., Marton F.: Far voxels: a multiresolution framework for interactive rendering of huge complex 3d models on commodity graphics platforms. *ACM Trans. Graph. 24*, 3 (2005), 878–885.

[GPMG10] Galin E., Peytavie A., Marchal N., Guérin E.: Procedural generation of roads. In *Computer Graphics Forum: Proceedings of Eurographics* (2010), vol. 29.

[GS02] Garland M., Shaffer E.: A multiphase approach to efficient surface simplification. In *Proceedings of the conference on Visualization '02* (2002), pp. 117–124.

[GSS99] Guskov I., Sweldens W., Schröder P.: Multiresolution signal processing for meshes. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques* (1999), SIGGRAPH '99, pp. 325–334.

[HGA*10] Hnaidi H., Guérin E., Akkouche S., Peytavie A., Galin E.: Feature based terrain generation using diffusion equation. *Computer Graphics Forum 29*, 7 (2010), 2179–2186.

[HGB93] Heiden W., Goetze T., Brickmann J.: Fast generation of molecular surfaces from 3d data fields with an enhanced marching cube algorithm. *Journal of Computational Chemistry 14*, 2 (1993), 246–250.

[Hop96] Hoppe H.: Progressive meshes. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques* (1996), pp. 99–108.

[Hop97] Hoppe H.: View-dependent refinement of progressive meshes. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques* (1997), pp. 189–198.

[Hop98] Hoppe H.: Efficient implementation of progressive meshes. *Computers & Graphics 22*, 1 (1998), 27–36.

[HSH09] Hu L., Sander P. V., Hoppe H.: Parallel view-dependent refinement of progressive meshes. In *I3D '09: Proceedings of the 2009 symposium on Interactive 3D graphics and games* (2009), pp. 169–176.

[Huf52] Huffman D.: A method for the construction of minimum redundancy codes. *Proceedings of the IRE 40*, 9 (Sep. 1952), 1098–1101.

[JC06] Johansson G., Carr H.: Accelerating marching cubes with graphics hardware. In *Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research* (New York, NY, USA, 2006), CASCON '06, ACM.

[JGA09] Jamin C., Gandoin P.-M., Akkouche S.: Technical section: Chumi viewer: Compressive huge mesh interactive viewer. *Comput. Graph. 33*, 4 (August 2009), 542–553.

[JT98] Jas A., Touba N. A.: Test vector decompression via cyclical scan chains and its application to testing core-based designs. In *ITC'98* (1998), pp. 458–464.

[JWLL06] Ji J., Wu E., Li S., Liu X.: View-dependent refinement of multiresolution meshes using programmable graphics hardware. *Vis. Comput. 22*, 6 (2006), 424–433.

[KBKv09] Kristof P., Beneš B., Krivanek J., Šťava O.: Hydraulic erosion using smoothed particle hydrodynamics. In *Proceedings of Eurographics 2009: Computer Graphics Forum 28 (2)* (2009).

[KCL06] Kim J., Choe S., Lee S.: Multiresolution random accessible mesh compression. *Computer Graphics Forum 25*, 3 (2006), 323–332.

[KCVS98] Kobbelt L., Campagna S., Vorsatz J., Seidel H.-P.: Interactive multi-resolution modeling on arbitrary meshes. In *SIGGRAPH* (1998), pp. 105–114.

[KL01] Kim J., Lee S.: Truly selective refinement of progressive meshes. In *Graphics Interface 2001* (2001), pp. 101–110.

[KMN88] Kelley A. D., Malin M. C., Nielson G. M.: Terrain simulation using a model of stream erosion. In *SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques* (1988), pp. 263–268.

[KVS99] Kobbelt L., Vorsatz J., Seidel H.-P.: Multiresolution hierarchies on unstructured triangle meshes. *Comput. Geom. 14*, 1-3 (1999), 5–24.

[KWP*03] Knieser M. J., Wolff F. G., Papachristou C. A., Weyer D. J., McIntyre D. R.: A technique for high ratio lzw compression. In *Proceedings of the conference on Design, Automation and Test in Europe - Volume 1* (Washington, DC, USA, 2003), DATE '03, IEEE Computer Society, pp. 10116–.

[LC87] Lorensen W. E., Cline H. E.: Marching cubes: A high resolution 3d surface construction algorithm. *ACM SIGGRAPH Computer Graphics 21*, 4 (1987), 163–169.

[Lev02] Levenberg J.: Fast view-dependent level-of-detail rendering using cached geometry. In *VIS '02: Proceedings of the conference on Visualization '02* (2002), pp. 259–266.

[LF03] Le Feuvre L.: Modelling and deformation of surfaces defined over finite elements. In *Proceedings of the Shape Modeling International* (2003), IEEE Computer Society, p. 175.

[Lin00] Lindstrom P.: Out-of-core simplification of large polygonal models. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques* (2000), SIGGRAPH '00, pp. 259–262.

[Liv99] Livnat Y.: *NOISE, WISE and SAGE: Algorithms for Rapid Isosurface Extraction.* PhD thesis, Univeristy of Utah, December 1999.

[Los04] Losasso F.: Geometry clipmaps: terrain rendering using nested regular grids. *ACM Transactions on Graphics 23* (2004), 769–776.

[LSCo*04] Lipman Y., Sorkine O., Cohen-or D., Levin D., Rössl C., peter Seidel H.: Differential coordinates for interactive mesh editing. In *In Proceedings of Shape Modeling International* (2004), Society Press, pp. 181–190.

[LSWW11] Lipp M., Scherzer D., Wonka P., Wimmer M.: Interactive modeling of city layouts using layers of procedural content. *Computer Graphics Forum (EG 2011) 30*, 2 (Apr 2011), 345–354.

[LT97] Low K.-L., Tan T.-S.: Model simplification using vertex-clustering. In *Proceedings of the 1997 symposium on Interactive 3D graphics* (1997), pp. 75–ff.

[LT98] Lindstrom P., Turk G.: Fast and memory efficient polygonal simplification. In *IEEE Visualization* (1998), pp. 279–286.

[Lue01] Luebke D. P.: A developer's survey of polygonal simplification algorithms. *IEEE Comp. Graph. Appl. 21* (2001), 24–35.

[Man83] Mandelbrot B. B.: *The fractal geometry of nature.* Freeman and Company, New York, 1983.

[Man99] Manzini G.: The burrows-wheeler transform: Theory and practice. In *Lecture Notes in Computer Science* (1999), Springer, pp. 34–47.

[Mar79] Martin G. N. N.: Range encoding: an algorithm for removing redundancy from a digitised message. In *In Video and Data Recording Conference* (July 1979).

[MBH*01] Meissner M., Bartz D., Hüttner T., Müller G., Einighammer J.: Generation of decomposition hierarchies for efficient occlusion culling of large polygonal models. In *Vision, Modeling, and Visualization* (2001), pp. 225–232.

[MBK07] Marinov M., Botsch M., Kobbelt L.: Gpu-based multiresolution deformation using approximate normal field reconstruction. *journal of graphics, gpu, and game tools 12*, 1 (2007), 27–46.

[MKM89] Musgrave F. K., Kolb C. E., Mace R. S.: The synthesis and rendering of eroded fractal terrains. *SIGGRAPH Computer Graphics 23*, 3 (1989), 41–50.

[MS91] Mueller H., Stark M.: *Adaptive Generation of Surfaces in Volume Data*. Tech. rep., Albert-Ludwigs University at Freiburg, 1991.

[MSS94] Montani C., Scateni R., Scopigno R.: A modified look-up table for implicit disambiguation of marching cubes. *The Visual Computer 10*, 6 (Dec. 1994), 353–355.

[MVN68] Mandelbrot B. B., Van Ness J. W.: Fractional brownian motions, fractional noises and applications. *SIAM Review 10*, 4 (1968), 422–437.

[NVI10] NVIDIA: *GF100 Processor Architecture*, 2010.

[NVI11a] NVIDIA: *CUDA C BEST PRACTICES GUIDE*. Design Guide Version 4.1, NVIDIA Corporation, 2011.

[NVI11b] NVIDIA: *NVIDIA CUDA*. C Programming Guide Version 4.1, NVIDIA Corporation, 2011.

[Paj01] Pajarola R.: Fastmesh: Efficient view-dependent meshing. In *9th Pacific Conference on Computer Graphics and Applications* (2001), pp. 20–30.

[Pau94] Paul Bourke: Polygonising a scalar field: Also known as: 3d contouring, marching cubes, surface reconstruction, May 1994.

[PD04] Pajarola R., DeCoro C.: Efficient implementation of real-time view-dependent multiresolution meshing. *IEEE Transactions on Visualization and Computer Graphics 10*, 3 (2004), 353–368.

[Per85] Perlin K.: An image synthesizer. *SIGGRAPH Comput. Graph. 19*, 3 (July 1985), 287–296.

[Per02] Perlin K.: Improving noise. *ACM Trans. Graph. 21*, 3 (July 2002), 681–682.

[PGTG04] Pouderoux J., Gonzato J.-C., Tobor I., Guitton P.: Adaptive hierarchical rbf interpolation for creating smooth digital elevation models. In *GIS '04: Proceedings of the 12th annual ACM international workshop on Geographic information systems* (2004), pp. 232–240.

[PH93] Prusinkiewicz P., Hammel M.: A fractal model of mountains with rivers. In *Graphics Interface '93* (1993), pp. 174–180.

[PH97] Popović J., Hoppe H.: Progressive simplicial complexes. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques* (1997), SIGGRAPH '97, pp. 217–224.

[PR00] Pajarola R., Rossignac J.: Compressed progressive meshes. *IEEE Transactions on Visualization and Computer Graphics 6*, 1 (2000), 79–93.

[RB93] Rossignac J., Borel P.: Multi-resolution 3d approximations for rendering complex scenes. In *Modeling in Computer Graphics: Methods and Applications* (Berlin, June 1993), Falcidieno B., Kunii T., (Eds.), Springer Verlag, pp. 455–465.

[RDG*04] Reck F., Dachsbacher C., Grosso R., Greiner G., Stamminger M.: Realtime isosurface extraction with graphics hardware. In *Eurographics 2004, Short Presentations and Interactive Demos* (2004), pp. 33–36.

[Roe] Roettger S.: The volume library. `http://www9.informatik.uni-erlangen.de/External/vollib/` (31.05.2012).

[Sai04] Said A.: *Introduction to Arithmetic Coding Theory and Practice*. Tech. Rep. HPL-2004-76, Hewlett-Packard Laboratories, 2004.

[SBBK08] Stava O., Benes B., Brisbin M., Krivanek J.: Interactive terrain modeling using hydraulic erosion. In *Eurographics/SIGGRAPH Symposium on Computer Animation* (2008), Gross M., James D., (Eds.), pp. 201–210.

[SCOL*04] Sorkine O., Cohen-Or D., Lipman Y., Alexa M., Rössl C., Seidel H.-P.: Laplacian surface editing. In *Proceedings of the 2004 Eurographics/ACM SIGGRAPH symposium on Geometry processing* (New York, NY, USA, 2004), SGP '04, ACM, pp. 175–184.

[SG01] Shaffer E., Garland M.: Efficient adaptive simplification of massive meshes. In *Proceedings of the conference on Visualization '01* (2001), pp. 127–134.

[SHLJ96] Shen H.-W., Hansen C. D., Livnat Y., Johnson C. R.: Isosurfacing in span space with utmost efficiency (issue). In *VIS '96: Proceedings of the 7th conference on Visualization '96* (Los Alamitos, CA, USA, 28-29 October 1996), IEEE Computer Society Press, pp. 287–ff.

[SHZO07] Sengupta S., Harris M., Zhang Y., Owens J. D.: Scan primitives for gpu computing. In *Graphics Hardware 2007* (2007), pp. 97–106.

[SM06] Sander P. V., Mitchell J. L.: Progressive buffers: view-dependent geometry and texture lod rendering. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Courses* (2006), pp. 1–18.

[SS05] Stachniak S., Stuerzlinger W.: An algorithm for automated fractal terrain deformation. In *Proceedings of Computer Graphics and Artificial Intelligence* (2005), pp. 64–76.

[ST89] Szeliski R., Terzopoulos D.: From splines to fractals. *SIGGRAPH Computer Graphics 23* (1989), 51–60.

[SW03] Schaefer S., Warren J.: Adaptive vertex clustering using octrees. In *SIAM Geometric Design and Computing* (2003).

[SW04] Schaefer S., Warren J.: Dual marching cubes: Primal contouring of dual grids. In *PG '04: Proceedings of the Computer Graphics and Applications, 12th Pacific Conference* (Washington, DC, USA, 6-8 October 2004), IEEE Computer Society, pp. 70–76.

[Tau95] Taubin G.: A signal processing approach to fair surface design. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques* (1995), SIGGRAPH '95, ACM, pp. 351–358.

[TPG99] Treece G. M., Prager R. W., Gee A. H.: Regularised marching tetrahedra: improved iso-surface extraction. *Computers and Graphics 23*, 4 (1999), 583–598.

[TR99] Taubin G., Rossignac J.: 3d geometry compression. In *SIGGRAPH '99: Course Notes*. ACM, Los Angeles, Aug. 1999. Course 22.

[TSD07] Tatarchuk N., Shopf J., DeCoro C.: Real-time isosurface extraction using the gpu programmable geometry pipeline. In *ACM SIGGRAPH 2007 courses* (New York, NY, USA, 2007), SIGGRAPH '07, ACM, pp. 122–137.

[UDG12] Ulrich C., Derzapf E., Guthe M.: *Parallel Out-of-Core Iso-Surface Extraction and Simplification*. Tech. rep., Philipps-Universität Marburg, Fachbereich 12 - Mathematik und Informatik, Arbeitsgruppe Grafik und Multimedia Programmierung, 2012.

[WG90] Wilhelms J., Gelder A. V.: Octrees for faster isosurface generation. *ACM SIGGRAPH Computer Graphics 24*, 5 (November 1990), 57–62.

[WNC87] Witten I. H., Neal R. M., Cleary J. G.: Arithmetic coding for data compression. *Commun. ACM 30* (June 1987), 520–540.

[WW94] Welch W., Witkin A.: Free-form shape design using triangulated surfaces. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques* (1994), SIGGRAPH '94, ACM, pp. 247–256.

[XV96] Xia J. C., Varshney A.: Dynamic view-dependent simplification for polygonal models. In *VIS '96: Proceedings of the 7th conference on Visualization '96* (1996), p. 327 ff.

[YL07] Yoon S.-e., Lindstrom P.: Random-accessible compressed triangle meshes. *IEEE Transactions on Visualization and Computer Graphics 13*, 6 (November 2007), 1536–1543.

[YSGM04] Yoon S.-E., Salomon B., Gayle R., Manocha D.: Quick-vdr: Interactive view-dependent rendering of massive models. In *VIS '04: Proceedings of the conference on Visualization '04* (2004), pp. 131–138.

[ZLS08] Zhang J., Long X., Suel T.: Performance of compressed inverted list caching in search engines. In *Proceedings of the 17th international conference on World Wide Web* (2008), WWW '08, ACM, pp. 387–396.

[ZSS97] Zorin D., Schröder P., Sweldens W.: Interactive multiresolution mesh editing. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques* (1997), SIGGRAPH '97, pp. 259–268.

[ZSTR07] Zhou H., Sun J., Turk G., Rehg J. M.: Terrain synthesis from digital elevation models. *IEEE Transactions on Visualization and Computer Graphics 13*, 4 (2007), 834–848.

Publications

Title	Authors	Year	Journal
Parallel View-Dependent Refinement of Compact Progressive Meshes	E. Derzapf, N. Menzel, M. Guthe	2010	Eurographics Symposium on Parallel Graphics and Visualization 2010 (EGPGV10)
Parallel View-Dependent Out-of-Core Progressive Meshes	E. Derzapf, N. Menzel, M. Guthe	2010	Vision Modeling and Visualization 2010 (VMV2010)
River Networks for Instant Procedural Planets	E. Derzapf, B. Ganster, M. Guthe, R. Klein	2011	In Computer Graphics Forum 2011 (Pacific Graphics 2011)
Instant Level-of-Detail	N. Grund, E. Derzapf, M. Guthe	2011	Vision Modeling and Visualization 2011 (VMV2011)
Dependency Free Parallel Progressive Meshes	E. Derzapf, M. Guthe	2012	In Computer Graphics Forum 2012
Parallel Progressive Mesh Editing	E. Derzapf, N. Grund, M. Guthe	2012	Unpublished
Parallel Out-of-Core Iso-Surface Extraction and Simplification	C. Ulrich, E. Derzapf, M. Guthe	2012	Unpublished

LEBENSLAUF

Dipl. Inf. Evgenij Derzapf

Anschrift: Erlenweg 35
35321 Laubach
Deutschland

Telefon: 06405/950210
E-Mail: derzapf@mathematik.uni-marburg.de

Geburtsdatum: 03.02.1984
Geburtsort: Mirnij

Familienstand: verheiratet

Ausbildung / Studium

Seit 2010	**Philipps-Universität Marburg** **Fachbereich Mathematik und Informatik - AG Grafik und Multimedia** **Doktorand**

2004 – 2010	**Philipps-Universität Marburg**
	Hauptfach Informatik mit den Nebenfächer Physik (Grundstudium) und Mathematik
	▪ Thema der Diplomarbeit: „Progressive Meshes mit CUDA“
	▪ Abschluss: Diplom-Informatiker
	▪ Schwerpunkte:
	  ▪ Grafikprogrammierung
	  ▪ (Modellgetriebene) Softwaretechnik/-entwicklung
	  ▪ Mathematik (Schwerpunkte: Numerik und Kryptologie)

2000 – 2003	**Herderschule Giessen** ▪ Abschluss: Abitur
1998 – 2000	**Friedrich-Magnus-Gesamtschule Laubach** ▪ Abschluss: Realschule
1991 – 1998	**Gymnasialschule in Sankt-Petersburg**

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich meine Dissertation

Parallel Mesh Processing

selbständig, ohne unerlaubte Hilfe angefertigt und mich dabei keiner anderen als der von mir ausdrücklich gekennzeichneten Quellen und Hilfen bedient habe.

Einer Doktorprüfung habe ich mich bisher nicht unterzogen. Die Dissertation wurde in der jetzigen oder einer ähnlichen Form noch bei keiner anderen Hochschule eingereicht und hat noch keinen sonstigen Prüfungszwecken gedient.

Marburg, 22. Oktober 2012